

Achs Ágnes–Szendrői Etelka

**Programozás 2., I. kötet
Az objektumorientált paradigma alapjai**

Pécs
2015

A tananyag a TÁMOP-4.1.1.F-14/1/KONV-2015-0009 azonosító számú,
„A gépészeti és informatikai ágazatok duális és moduláris képzéseinek kialakítása a
Pécsi Tudományegyetemen” című projekt keretében valósul meg.



Programozás 2., I. kötet

Achs Ágnes-Szendrői Etelka

Szakmai lektor: Szabó Levente

Nyelvi lektor: Veres Mária

Kiadó neve

Kiadó címe

Felelős kiadó:

ISBN szám

Pécsi Tudományegyetem
Műszaki és Informatikai Kar

Pécs, 2015
© Achs Ágnes, Szendrői Etelka



TARTALOMJEGYZÉK

1.	Bevezetés.....	1
1.1.	Objektumorientált programozás (<i>Achs Á.</i>).....	2
1.2.	.Net architektúra (<i>Szendrői E.</i>)	8
1.2.1.	A Common Language Runtime (CLR).....	10
1.2.2.	A .NET alapú alkalmazás fordítása, végrehajtása	11
1.2.3.	A .NET Framework fejlődése	11
1.2.4.	A Common Type System.....	13
1.3.	A C# programozási nyelv (<i>Szendrői E.</i>)	14
1.3.1.	A C# nyelv szintaktikája.....	15
1.3.2.	A C# program szerkezete.....	28
2.	Az objektumorientáltság alapjai	31
2.1.	Elméleti háttér (<i>Szendrői E.</i>).....	31
2.2.	Gyakorlati példa (<i>Achs Á.</i>)	50
3.	A C# listakezelése	60
3.1.	Elméleti háttér (<i>Szendrői E.</i>).....	60
3.1.1.	Tömbök kezelése	60
3.1.2.	Gyűjtemények, listák kezelése	65
3.1.3.	Listák feltöltése fájlból, fájlba írás	69
3.2.	Gyakorlati példa (<i>Achs Á.</i>)	71
4.	Öröklődés, láthatóság	81
4.1.	Elméleti háttér (<i>Szendrői E.</i>).....	81
4.1.1.	Az öröklődés megvalósítása C# nyelven.....	82
4.1.2.	Virtuális metódusok	86



4.1.3.	Láthatóság, védelem	87
4.1.4.	Absztrakt osztályok.....	87
4.1.5.	Lezárt osztályok és lezárt metódusok	88
4.1.6.	Interfészlek	88
4.2.	Gyakorlati példák (<i>Achs Á.</i>)	92
4.2.7.	Állatverseny folytatása	92
4.2.8.	Járműpark.....	105



TÁBLÁZATOK JEGYZÉKE

1. táblázat A C# beépített típusai	15
2. táblázat C# kulcsszavak	16
3. táblázat Aritmetikai operátorok	17
4. táblázat C# hozzáférési módosítók.....	23
5. táblázat Metódus paramétertípusok	26



ÁBRÁK JEGYZÉKE

1. ábra A .NET Framework.....	8
2. ábra Alkalmazás fordítása, végrehajtása.....	11
3. ábra A .NET Framework fejlődése	12
4. ábra A C# alaptípusai.....	14
5. ábra A C# program szerkezete	28
6. ábra A Visual Studio.....	29
7. ábra A Visual Studio munkaablakai	30
8. ábra A Lakas osztály diagramja	33
9. ábra Objektum létrehozása	34
10. ábra Objektum létrehozása paraméteres konstruktorral	36
11. ábra Statikus és példányadattagok	38
12. ábra A System.Array osztály néhány statikus metódusa	63
13. ábra Öröklődés UML diagram	81



1. Bevezetés

*Mondd el, és elfelejtem;
mutasd meg, és megjegyzem;
engedd, hogy csináljam, és megértem.*

Konfuciusz

Kedves Olvasó!

Ez a jegyzet a Pécsi Tudományegyetem Műszaki és Informatika karán tanított Programozás 2. tárgyhoz készült, de nagy örömkre szolgálna, ha más is szívesen tanulna belőle.

Ahogy láthatja is a tartalomjegyzékből, a jegyzet az objektumorientált gondolkozás és a C# nyelv rejtelmeibe próbálja meg bevezetni a tisztelt olvasót. Az elméleti alapokat minden egyes fejezetben egy vagy több gyakorlati példa és azok magyarázatokkal ellátott lehetséges megoldása követi.

A mottóként szolgáló idézet programozásra legalább annyira érvényes, mint bármi másra. Programozni csak akkor lehet megtanulni, ha az ember saját maga írja a programokat. Ha kedve van, akár önállóan is nekiállhat bármelyik feladatnak, de utána – vagy a program megírása előtt – érdemes átolvasni a javasolt megoldást is. Hogy semmiképpen ne érezzen csábító kísértést, a programrészletek képként, azaz nem másolható módon vannak beilleszтve.

A közölt megoldások nem sok előismeretet feltételeznek, de valamennyit azért igen, ezeket persze menet közben is meg lehet szerezni. Feltételezzük, hogy Önnek már van valamennyi algoritmikus programozási ismerete. A megírandó programok feltételezik, hogy az olvasó más nyelven írt már ciklust, elágazást, ismeri a tömbököt stb.

A mintapéldák a Visual Studio fejlesztőkörnyezetében készültek, és feltételezzük, hogy Ön, kedves Olvasó, szintén ismeri ezt a fejlesztőeszközt, vagy hamarosan megismeri. Ezért a fejlesztőkörnyezet által generált kódok jó részét nem közli a megoldás.

Kívánjuk, kedves Olvasó, hogy élvezettel tanulja az itt leírtakat, és találjon sok-sok örömet a feladatok megoldásában!

Pécs, 2015. szeptember



Achs Ágnes



Szendrői Etelka

1.1. Objektumorientált programozás (Achs Á.)

Az élet szép, környezetünk tele van fákkal, virágokkal, repdeső madarakkal, vidáman futkározó állatokkal.



Ez a – nem művészzi értékű, de idillikus – kép azt a pillanatot mutatja, amikor még nincs ott az ember. Ha ő is megérkezik, akkor jó esetben gyönyörködik benne, de egy kis idő után igényét érzi annak, hogy valakinek meséljen a látottakról. Ehhez viszont meg kell alkotnia a fa, virág, madár, kutya, macska stb. fogalmát. De hogyan érti meg a hallgatósa, hogy mire gondol, mit láthatott, amikor elmeséli az élményeit? Csak akkor tudják elképzelni a hallottakat, ha bennük is élnek ezek a fogalmak, és maguk is „látják” a képet. Persze, nem ugyanazt fogják látni, mint a mesélő, sőt a hallgatóság minden tagja mászt és mást képzeli el, de mégis megtalálják a közös hangot, mert jó esetben a fogalmaik lényege azonos: tudják, hogy a fa olyan valami, amelynek „földbe gyökerezett a lába”, a szél hatására hajlongani tud, az állatok viszont változtathatják a helyüket, a madarak repülni tudnak, a kutyák ugatni stb.

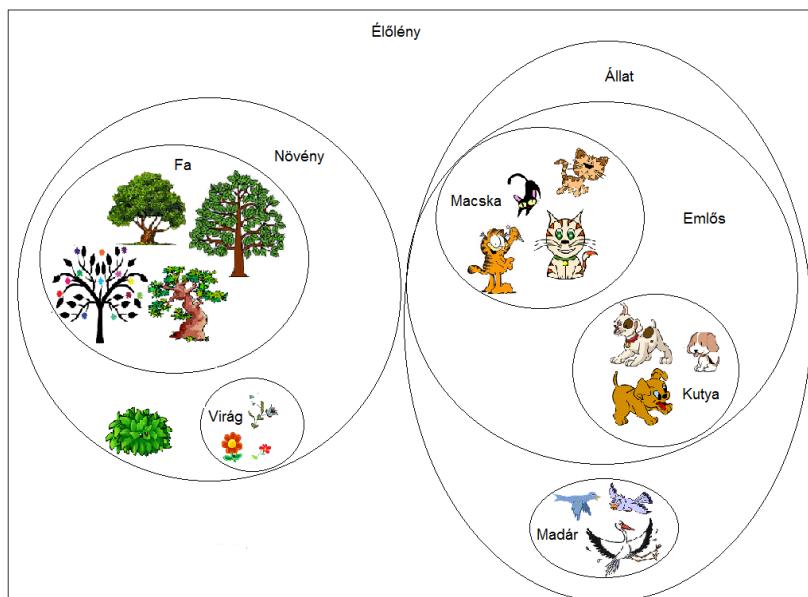
Ha viszont le kell fényképezni egy adott fát, vagy hazahozni a rétről a család kutyáját, akkor már nem elég fogalmi szinten gondolkozni, hanem a konkrét „példánnyal” kell foglalkoznunk.

De hogyan alakulhattak ki ezek a közös fogalmak, illetve hogyan alakulhatnak ki újabbak? Úgy, hogy állandóan osztályozzuk a látottakat. A kép szereplőinél maradva, először talán csak annyit veszünk észre, hogy mindegyikük él, aztán azt, hogy vannak köztük egy helyben maradó, illetve a helyüket változtató élőlények, később esetleg még további megkülönböztető jegyeket fedezünk fel, és finomítjuk a fogalmainkat. (A kisgyerek is hasonlóan fedez fel a világot, bár ez a felfedezés kétirányú, eleinte inkább a konkrét ismeretek felől indulva jutunk el elvontabb szintig, majd a már meglévő tudásunkba kell integrálni az újdonságokat. De maga a fogalomalkotás hasonló az itt tárgyaltakhoz.)

Térjünk vissza a képen látottakhoz. Észrevettük tehát, hogy csupa élőlény látható rajta. Aztán megkülönböztettük a helyváltoztatásra képes és képtelen lényeket. Az előbbiekt az állatok, az utóbbiak a növények. De még további lényegi különbségeket is észrevehetünk: a

növények között vannak sokáig élő fás szárúak (*fa*), illetve rövid életű lágy szárúak (*virág*). Az állatok egy része röpdös a levegőben (*madár*), más részük a földön szaladgál, és így tovább. Amikor azt tapasztaljuk, hogy lényegi eltérés van a vizsgált élőlények között, akkor külön osztályba soroljuk őket, ha azt észleljük, hogy bár van eltérés, de sokkal fontosabbak a közös jellemzők, akkor azonos osztályba kerülnek. Ez a fajta csoportosítási, osztályozási képesség alapvető része az emberi gondolkozásnak, és ugyanez az alapja az **objektumorientált** gondolkozásmódnak is.

A csoportokba (osztályokba) sorolás hatására létrejöhét a következő fogalomhierarchia, vagy más néven, osztályhierarchia:



Ahogy látható, különböző kapcsolatok vannak az egyes csoportok között. Mindegyik benne van az élőlényeket szimbolizáló téglalapban, de vannak egymástól független csoportok, illetve vannak olyanok is, ahol az egyik tartalmazza a másikat. Például a kutya csoportja az emlősök csoportján belülre van rajzolva, az pedig az állatok csoportján belülre. Ez a tartalmazás logikus, hiszen egy kutya egyúttal emlős is, és minden emlős az állatok csoportjába tartozik. Ezért amikor a kutya fogalmát akarjuk meghatározni, vagyis azokat a jellemzőket, amelyek alapján egy élőlényt kutynak tekintünk, akkor elég csak a specialitásokat kiemelni anélkül, hogy az emlősökre, illetve az állatokra vonatkozó egyéb tulajdonságokat külön részletezni kellene. Az ilyen tartalmazási relációt (vagyis azt, amikor közöljük, hogy a kutya egyúttal emlős is, azaz a kutya fogalma az emlős fogalmának kibővítése) **öröklődésnek** (esetleg származtatásnak vagy kibővítésnek) nevezzük.

De mi van az egyes csoportokon belül? Szemmel láthatóan a kezdőkép konkrét élőlényei. Vagyis az osztályozás mindig kétirányú:

Az egyik irány az **absztrakció**. Ennek során megpróbáljuk kiemelni az azonos osztályba került dolgok közös jellemzőit: megtartjuk a lényegesnek vélt tulajdonságokat, és elhagyjuk a lényegteleneket.

A másik irány: a kialakult osztályok használata, vagyis ha definiáltunk egy osztályt, akkor hogyan lehet olyan példányokat létrehozni, amelyek ehhez az osztályhoz tartoznak. (Esetünkben: ha pl. bemegyünk egy kertészetbe fát vásárolni, akkor valóban fát kapunk.)

Az **osztály** tehát egy absztrakt fogalom (amolyan tervrajzféle), az osztályba sorolt konkrét dolgok pedig az osztály **példányai**, vagy más szóval **objektumok**.

Programozási szempontból azt is mondhatjuk, hogy az osztály egy összetett típust jelent, ahol mi magunk (vagy az adott programnyelv létrehozói) definiáljuk azt, hogy mit is értünk ez alatt a típus alatt, az objektumok pedig ilyen típusú változók.

Alapfogalmak:

A valós világ objektumainak kétféle jellemzője van: mindegyiknek van valamilyen állapota (valamilyen tulajdonsága), és mindegyik viselkedik valamilyen módon. Például egy kutya tulajdonsága lehet a neve, színe, fajtája; viselkedése pedig az, hogy ugat, csóválja a farkát stb. Az osztályozás során pontosan ezeket a tulajdonságokat és viselkedést kell leírnunk, illetve meghatároznunk.

Az **objektumorientált programozás (OOP)** egy, az osztályhierarchiára épülő programozási módszer, amely lehetővé teszi különböző bonyolult változók (objektumok) létrehozását és kezelését. Egy **objektumorientált program** az egymással kapcsolatot tartó, együttműködő objektumok összessége, ahol minden objektumnak megvan a jól meghatározott feladata.

Az **osztály** egy-egy fogalom definiálására szolgál. Leírásakor egy-egy speciális típust határozunk meg abból a célból, hogy később ilyen típusú változókkal tudunk dolgozni. Az Osztaly típusú változó majd Osztaly típusú objektumot tartalmaz. Egy osztály tulajdonképpen egy objektum „tervrajzának” vagy sémájának tekinthető.

A **példány** egy konkrét, az osztályra jellemző tulajdonságokkal és viselkedéssel rendelkező **objektum**. Mindkét elnevezés használatos (példány, objektum). Egy időben több azonos típusú objektum is lehet a memoriában, és két objektumot akkor is különbözőnek tekintünk, ha azonos tulajdonságaik vannak. (Pl. két Bodri nevű puli nyilván két különböző kutya.)

Egy program objektumai hasonlóak a valós világ objektumaihoz, vagyis nekik is vannak állapotaik (tulajdonságaik) és viselkedésük. Ezeket az állapotokat úgynevezett mezőkben (vagy adattagokban) tároljuk, a viselkedést pedig a metódusok írják le. Mivel az azonos osztályba tartozók hasonló módon viselkednek, ezért a hozzájuk tartozó metódusokat az osztályok definiálásakor határozzuk meg. Azt is, hogy milyen mezőkkel kell rendelkeznie egy-egy ilyen osztálynak (azaz ilyen típusnak), de a mezők konkrét értékét már az objektumok, azaz a konkrét példányok tartalmazzák.

De hogyan jönnek létre ezek a példányok? A **konstruktur** hozza létre őket. Ez egy speciális, visszatérési típus nélküli metódus, amelyben inicializáljuk az objektum bizonyos állapotait, és helyet foglalunk számára a memoriában. Az, hogy helyet kapnak a memoriában, azt jelenti, hogy minden egyes példány az összes adattagjával együtt helyet kap. Egy kivétel lehet, amikor minden egyes példányhoz azonos értékű adattag tartozik. (Például minden magyar állampolgár 18 éves korában válik választópolgárrá.) Az ilyen adatot fölösleges annyi példányban tárolni, ahány objektum van, elég csak egyszer. Ezeket, az azonos típusú objek-

tumok által közösen használható adatokat, **statikus** adatoknak nevezzük, illetve a rájuk hivatkozó változókat statikus változóknak. Léteznek statikus metódusok is, ezeket az öket tartalmazó osztály példányosítása nélkül tudjuk meghívni.

Létrejöttük után az objektumok „önálló lények”, kommunikálni tudnak egymással. Bár ennél kicsit többet jelent a kommunikáció, de első közelítésben mondhatjuk azt, hogy gyakorlatilag azt jelenti, hogy az egyik objektum meg tudja hívni a másik valamelyik metódusát.

Ugyanakkor nem szabad megengednünk azt, hogy kívülről bárki belepiszkálhasson egy objektum állapotába, vagyis hogy egy objektum megváltoztathassa egy másik adattagjának értékét, illetve lekérhesse azt annak ellenére, hogy a másik esetleg titokban szeretné tartani. (Nem feltétlenül örül annak valaki, ha bárki megnézheti, mennyi pénz van a bankszámláján, és nyilván nem lehet kívülről megváltoztatni valaki születési dátumát.)

Azt az elvet, hogy egy objektumot ne lehessen kívülről nem várt módon manipulálni, az **egységbezárás** (vagy az **információ elrejtése**) elvének nevezzük. Ennek lényege, hogy csak meghatározott metódusokon keresztül módosítható az állapot. Erre mutat egy kis példát a mellékelt kép: a tanárnak nem feltétlenül kell tudnia, hogy a vizsgázó könyvekből vagy internet alapján készült fel. Őt csak az érdeklődik, hogy tud-e a diákok – azaz, visszafordítva az OOP nyelvére, hogy elvárt módon működik-e az objektum.



Az, hogy elvárt módon működik (vagyis pl. az előbb említett diákok jól felel), azt jelenti, hogy meg tudjuk hívni az objektum megfelelő metódusát, és az úgy működik, ahogyan kell. De ahhoz, hogy meg tudjuk hívni, a mezővel ellentétben, a metódus nem lehet rejtett. Lehetnek olyan metódusok, amelyekhez bárki hozzáférhet, de lehetnek olyanok is, amelyeket csak belső használatra szánunk, illetve olyanok is, amelyet csak bizonyos körülmények között akarunk megosztani. Azt, hogy ki érheti el a metódusokat, a **láthatóság** szabályozza.

Az egyes objektumorientált nyelvek között lehet eltérés, de az alapvető három láthatósági típus a nyilvános (**public**), rejtett (**private**) és a védett (**protected**) mód. Ezenkívül a default láthatóság, vagyis az, amikor nincs láthatósági módosító a metódusnév előtt. (Láthatósági módosítók lehetnek osztálynév és mezőnév előtt is, de ezekre, illetve a láthatóság pontosabb definiálására majd a megfelelő helyen és időben sor kerül.)



Ahogy az induló példában már szó volt róla, az osztályok között kapcsolat is lehet. Az egyik leggyakrabban használt kapcsolat a már említett **öröklődés**.

A minket körülvevő világban gyakran előfordul, hogy két tárgy (élőlény stb.) között hasonlóságot tapasztalunk. A képen látható zsiráfgyerekek is hasonlít a mamájára, sok-sok biológiai tulajdonságot örököl tőle. Ugyanakkor saját tulajdonságokkal (is) rendelkező, önálló egyéniséggel.

Hasonló a helyzet az egymással öröklési kapcsolatban lévő osztályokkal.

Programozási szempontból egyik alapvető elvárás, hogy a kódunkban lehetőleg ne legyen kódismétlés. Többek között ezt hivatott megoldani az öröklődés. Ha egy osztály egy másik osztály minden nyilvános tulajdonságát és metódusát tartalmazza, de vagy egy kicsit bővebb annál, vagy bizonyos metódusai kicsit eltérően működnek, mint a másik megfelelő metódusa, akkor ezt az osztályt célszerű származtatni (örökíteni) a másikból, és csak az eltérő tulajdonságait, metódusait tárolni, a közösekre pedig hivatkozni. Azt az osztályt, amelyet alapul veszünk, szülő- vagy ősosztálynak nevezzük, azt, amelyik kibővíti ezt, utód- vagy származtatott osztálynak.

Természetesen olyan is lehet, hogy egy osztály sok dolgot tartalmaz egy másik osztályból, de nem minden, illetve a közös tulajdonságokon kívül vannak még saját specialitásai is. Ekkor is alkalmazható az öröklődés, csak ekkor létre kell hoznunk egy közös ősosztályt, amelyből mindenki öröklőhet. Ha ez a közös ősosztály valóban csak arra kell, hogy minden osztály tudjon örökölni tőle, de nem akarunk saját példányokat létrehozni belőle, akkor célszerű **absztrakt**, azaz nem példányosítható **osztályként** definiálni. Ilyen lehet pl. a bevezető példa emlős osztálya, hiszen nincs egyetlen „csak” emlős példány sem, csak kutyák és macskák vannak. Egy absztrakt osztályban lehetnek absztrakt metódusok, vagyis olyanok, amelyeknek nem írjuk meg a törzsét. Esetünkben ilyen lehet például a „beszél” metódus, amelyet majd elég lesz a kutya, illetve macska osztályban kifejteni, hiszen a kutya ugatva „beszél”, a macska nyávogva.

Ennél komolyabb absztrakció is lehet, amikor csak azt soroljuk fel, hogy egyáltalán milyen metódusokat akarunk majd megvalósítani, de maga a megvalósítás hiányzik. Ezt nevezzük **interfésznek**, de ennek tárgyalására majd a megfelelő fejezetben kerül sor.

Az öröklődés kapcsán még egy fogalmat kell megemlítenünk, mégpedig a **polimorfizmus** fogalmát. A szó görög eredetű, és többalakúságot jelent. Ezt legegyszerűbben a már elkezdett példán tudjuk megvilágítani.

Mivel a kacsa nem emlős, ezért tekintsük a képen szereplő állatokat az Allat osztályból származtatott Kutya, Macska, Kacsa osztály egy-egy példányának. Az Allat osztályban megírt (esetleg absztrakt) beszel() metódust más-más módon írja felül az utód osztályok megfelelő metódusa. Ha a képen látható állatpéldányokat egy közös listában szeretnénk szerepeltetni, akkor kénytelenek leszünk Allat típusúnak deklarálni őket. Fordítási időben nem derül ki, hogy az adott lista esetében melyik beszel() metódust kell futtatni, futásidőben azonban – vagyis amikor kiderül, hogy melyik utód-osztályba tartozik a konkrét példány – ez egyértelművé válik.

Azt, hogy ilyen későn (vagyis csak futási időben, amikor sorra kerül az adott példány) dől el, hogy melyik metódust kell futtatni, **késői kötésnek** nevezzük.



Összefoglalva: az objektumorientált programozás legfontosabb alapfogalmai az osztály és az objektum, legfontosabb alapelvei pedig az egységbézárás, öröklődés és polimorfizmus.

Van azonban még egy fontos alapelv, az újrahasznosíthatóság elve, amely persze nem csak az OOP programokra igaz. Vagyis az, hogy úgy írunk programot, hogy azt ne csak egyszer, egy adott szituációban tudjuk felhasználni.

Ennek elnevezésére még angol mozaikszó is született: WORA („Write once, run anywhere”) vagy WORE („Write once, run everywhere”).

Vagyis úgy írjuk meg a programjainkat, hogy annak elemeit néhány egyszerű cserével könnyedén fel lehessen használni egy másik szoftver létrehozásakor.

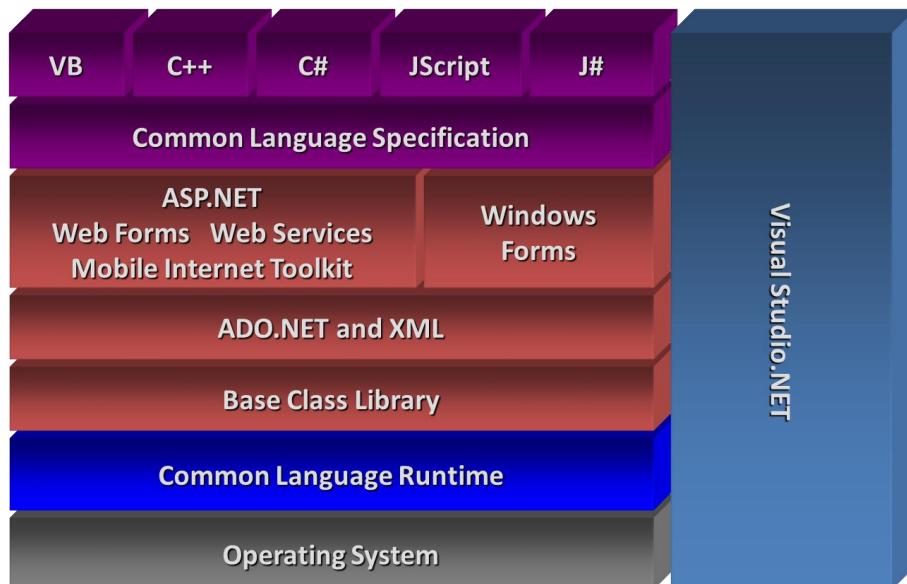
Ennek eléréséhez célszerű betartani a következő elveket:

- *Modularitás elve*: Próbáljuk meg a komplex feladatot kisebb részekre bontani, mégpedig úgy, hogy egy-egy rész egy-egy kisebb, önálló feladatot oldjon meg.
- *Fokozatos fejlesztés elve*: A fejlesztés elején előbb csak egy egyszerű implementációt hozzunk létre, teszteljük, majd bővítsük tovább, de úgy, hogy minden bővítési lépést tesztelünk.
- *Az adatreprézentáció rugalmasságának elve*: Az elv lényege, hogy bármikor könnyedén ki tudjuk cserélni a kód futásához használt adatokat. Ennek legelembb módja, hogy **SOHA** nem égetünk be adatokat. (De nem csak ezt jelenti az elv, eleve rugalmasan kell kezelni bármiféle adatot.)

Ez után a – zömében intuitív – bevezetés után jó munkát kívánunk az OOP gondolkozás elsajátításához, és kívánjuk, hogy ez a könyv hatékonyan segítse ezt a munkát.

1.2. .Net architektúra (*Szendrői E.*)

A Microsoft .NET különböző típusú és méretű alkalmazások fejlesztését és futtatását támogató infrastruktúra. Létrehozásának célja az volt, hogy az asztali számítógépen futó alkalmazásoktól a webes és mobil alkalmazásokon át a nagy információs rendszerekig egységes módon történjen a fejlesztés, minél hatékonyabban és gyorsabban. A .NET infrastruktúrának legfontosabb részei a .NET Framework, valamint a Visual Studio.NET. A keretrendszer összetevői, rétegei az 1. ábrán láthatók.



1. ábra A .NET Framework

A keretrendszer legfontosabb eleme a közös futtató környezet, a **CLR** (Common Language Runtime), amely közvetlenül az operációs rendszer szolgáltatásaira épül. Ez az összetevő biztosítja az alapvető futtatási szolgáltatásokat és gondoskodik a programok végrehajtásáról. A keretrendszer további összetevői a **Common Type System** (egységes típusrendszer) és a **Common Language Specification** (egységes nyelvi specifikáció).

A CLR felett helyezkedik el az alap osztálykönyvtár, a **Base Class Library**. Osztályokat, interfészket tartalmaz, amelyeket bármelyik .NET programozási nyelvből el lehet érni, fel lehet használni a készülő alkalmazásokban. Az osztályok funkciójuknak megfelelően csoportosítva, úgynevezett névterekbe vannak rendezve. A **System** névtér tartalmazza az alaposztályokat, amelyek gyakran használt adattípusokat, interfészket, eseményeket, attribútumokat stb. definiálnak. Itt található az összes osztály őse, az **Object** osztály is. Amikor egy programban például Windows formot akarunk használni, akkor a **System.Windows.Forms** névtérben található Form osztályra lesz szükségünk. A névterek más névtereket is tartalmazhatnak, így a **System** névtérből mint gyökérpontból kiindulva, fastruktúrát alkotnak. Egy adott ágon lévő színterek nevét ponttal választjuk el egymástól. Ez a logikai csoportosítás lehetővé teszi, hogy a fa más-más ágán lévő névterben ugyanolyan nevű osztály szerepeljen. Amikor egy Windows Formra elhelyezünk egy nyomógombot, akkor a **System.Windows.Forms** névtér **Button** osztályára van szükségünk. Ha

webalkalmazást készítünk, és a weblapon van szükségünk nyomógombra, akkor a *System.Web.UI.WebControls* névtérben található **Button** osztályt használjuk.

Az alap osztálykönyvtár felett van az **ADO.NET** és **XML** réteg, amely támogatja az adatelérést, adatmanipulációt és az XML generálást. Egységes adatkezelést biztosít, akár adatbázisban, akár XML fájlban vannak az adatok. Az ADO.NET szolgáltatásainak segítségével kapcsolódhatunk különböző adatbázisszerekhez, mint MS SQL Server, Oracle, IBM DB2, MySQL vagy ODBC kapcsolatokon keresztül más adatforrásokhoz.

Felfelé haladva az ábra rétegei között, a réteg kettéválik **ASP.NET** és **Windows Forms** rétegekre, amelyek web és vastag kliens (Windows Form) alkalmazások készítését támogatják. Az ASP.NET komponens teszi lehetővé, hogy webszervizeket használunk vagy hozzunk létre, valamint mobil eszközön működő alkalmazások létrehozására biztosít lehetőséget.

A **Common Language Specification (CLS)**, közös nyelvi specifikáció) réteg a .NET Frameworkben használható nyelvekre vonatkozó leírás. Azok a nyelvek, amelyek megfelelnek ennek a specifikációnak, alkalmasak a .NET alkalmazások fejlesztésére. A CLS definiálja a **Common Type System (CTS**, egységes típusrendszer) jelentős részét. Az egységes típusrendszer abban játszik szerepet, hogy a választott .Net nyelvtől függetlenül, a futtatási környezet egységesen kezelje az adatokat. Leírja az alaptípusok tárolási módját, méretét. Mivel a CTS flexibilis, sok nyelv adaptálható a .NET platformba, például Eiffel, Prolog stb.

A CLS felett helyezkedik el a programozási nyelvek és fordítóprogramjaik rétege. Az ábrán látható felsorolás nem teljes. A használható programozási nyelvek köre folyamatosan bővül. A legnépszerűbb programozási nyelv a .NET Környezetben a C# (ejtsd: szí sárp) és a Visual Basic.NET.

A Microsoft a .NET keretrendszerét és a C# nyelvet 2000-ben szabványosította, az ECMA keretein belül (<http://www.ecma-international.org/publications/standards/Ecma-334.htm>, és <http://www.ecma-international.org/publications/standards/Ecma-335.htm>). A szabványok utolsó módosítására 2006-ban (Ecma-334) és 2012-ben (Ecma-335) került sor.)

Több projekt indult a .NET Framework és a C# programozási nyelv más, nem Windows platformon történő alkalmazására. Az egyik ilyen sikeres projekt a Mono projekt, amely a .NET Framework nyílt forrású implementációját valósítja meg, a C# és a Common Language Runtime fent említett ECMA szabványára alapozva. (<http://www.mono-project.com>)

A .NET Frameworkben való alkalmazásfejlesztést támogatja a Visual Studio integrált fejlesztőeszköz. Grafikus felületen, számos szolgáltatást biztosít a fejlesztők számára mind a kódírás megkönnyítésében az intelligens szövegszerkesztőjével, mind a hibakeresés és a javítás területén a debug lehetőséggel. A Visual Studio különböző változatai, az ingyenes Express változattól kezdve, a Professional, illetve Ultimate kiadásokban elérhetők a fejlesztők számára. Ezekben a változatokban a program struktúrájának, az osztályok szerkezetének grafikus megjelenítésére, modellezésére is lehetőség van. Unit tesztek készítésére szintén támogatást ad. A Visual Studio Team System pedig a csoportos alkalmazásfejlesztést támogató, a Scrum módszertanra alapozott fejlesztést tesz lehetővé.

1.2.1. A Common Language Runtime (CLR)

A **Common Language Runtime** (CLR) a .NET Framework legalsó rétege maga is összetett, több komponensből áll, ezért részletesebben is foglalkoznunk kell vele. A CLR, a közös nyelvi futató rendszer biztosítja, hogy „felügyelt”, vagy angolul „managed” kód fut a keretrendszerben, s a programkód futása közben minden a memória-hozzáféréseket, minden a jogosultságokat ellenőrizni lehet.

A CLR különböző komponenseket foglal magába, amelyek a felügyelt kód végrehajtását támogatják. Egyik ilyen komponens a **Kivételkezelő**, amely a futás közbeni hibákat hivatott kezelni. A strukturált kivételkezelés egy objektumot alkalmaz a hibainformáció tárolására és külön kivételkezelő kódblokkot az objektum kezelésére.

A **Security Engine** (Biztonsági motor) a kódereedethez kötött biztonsági ellenőrzéseket hajtja végre. A programkód futása közben, az utasítások végrehajtása előtt ellenőrizni lehet a jogosultságot is, hogy az adott program és az adott felhasználó esetén, a soron következő utasítás végrehajtása megengedett-e (pl. kapcsolódhat-e egy adatbázishoz, írhat-e az adott fájlba vagy létesíthet-e hálózati kapcsolat).

A **Thread Support** (Szálkezelő) biztosítja a szálkezelést, valamint a futó szálak szinkronizálását. Támogatja a többszálú futást és felügyeli annak végrehajtását.

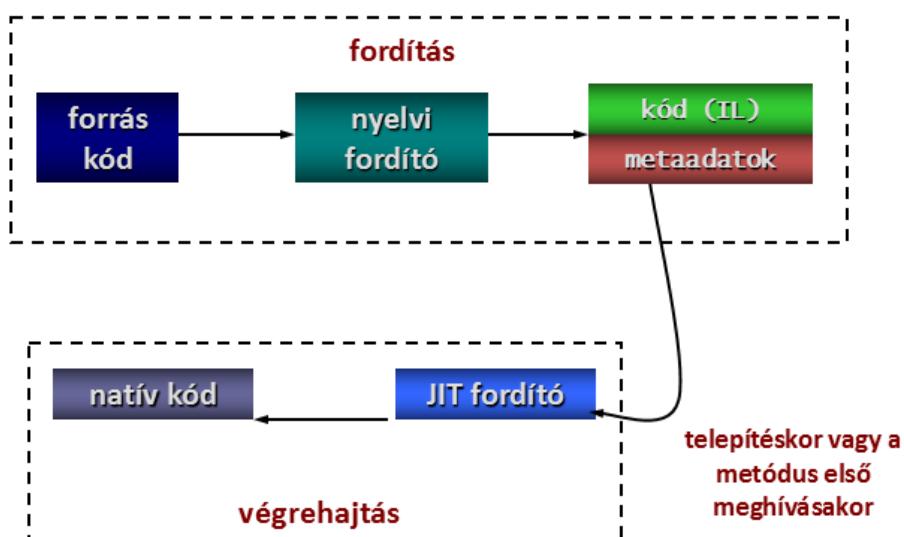
A **Type Checker** (Típusellenőrző) futási időben figyeli az egyes műveletek típushelyességét. Ez az a komponens, amely a .NET Frameworkben mindenhol megjelenő erős típusosságot futás közben is garantálja. Az erős típusosság azt jelenti, hogy minden művelet csak meghatározott hozzárendelt adattípusokkal végezhető. Ha egy műveletet olyan adaton próbálunk meg végrehajtani, amelynek a típusa nem felel meg az előírtaknak, akkor a CLR kivételkezelő mechanizmusa egy kivételt (exception) ad, és a nem megengedett műveletet nem lehet végrehajtani.

A CLR fontos szolgáltatása a memóriakezeléssel kapcsolatos automatikus szemétgyűjtés. Ezt a feladatot a **Garbage Collector** látja el, amely folyamatosan figyeli a futó programok memóriafelhasználását, s a programok által lefoglalt, de már nem használt memória felszabadítását végzi.

A CLR **Just-In-Time (JIT)** compiler feladata, hogy a forrásnyelvi fordítók által MSIL (Microsoft Intermediate Language), röviden IL köztes kódra fordított programot futás közben natív gépi kódú utasítások sorozatára fordítsa, a felhasználó CPU architektúrájának megfelelően. Ez a közös nyelvi infrastruktúra (Common Language Infrastructure, **CLI**) biztosítja a .NET nyelvfüggetlenségét. Elvileg bármilyen programozási nyelven készíthetünk alkalmazást, amelyhez létezik olyan fordítóprogram, amely ismeri a CTS és CLS követelményeit és képes a program forráskódját a CLR köztes kódjára, IL-re fordítani. Ha sikerült a köztes kódra fordítás, akkor a JIT fordító az adott gépi kódra lefordítja az utasításokat. Ez a mechanizmus lehetővé teszi azt is, hogy egy alkalmazás fejlesztése során az alkalmazás néhány programját egy bizonyos programozási nyelven, a többöt pedig egy másik programozási nyelven írjuk meg. Mivel minden nyelvi fordító ugyanarra a köztes nyelvre fordítja le a saját forráskódját, így a fordítás következtében az eredeti forrásnyelvek nyelvi különbözőségei teljesen eltűnnék és a befordított komponensek zavartalanul együttműködnek.

1.2.2. A .NET alapú alkalmazás fordítása, végrehajtása

Egy adott .NET nyelven megírt alkalmazást egy köztes nyelvre, az MSIL (röviden IL) nyelvre fordítjuk a nyelvi fordítóval. Az IL-kód közvetlenül nem futtatható, szükség van a JIT futásidejű fordításra. A .NET a lefordított, IL-utasításokat tartalmazó bináris fájljait **assemblynek** (szerelvénynék) nevezi. Az IL-utasítások mellett az assembly metaadatokat tartalmaz, amelyek a benne lévő objektumtípusok adatait, verzióinformációkat, a futás során használt egyéb objektumokra való hivatkozásokat, a biztonsági beállításokat stb. írják le. Az assembly lehet egyszerű .DLL vagy .EXE fájl. Amikor egy .NET alkalmazást indítunk, az indítás azt jelenti, hogy megkérjük a végrehajtási motort, töltse be az alkalmazásunk futtatásához szükséges komponensek moduljait. A végrehajtás minden esetben egy .NET osztály metódusa. A végrehajtás során a **Class Loader** (amely szintén a CLR egy komponense) gondoskodik arról, hogy a keresett osztályhoz tartozó assembly betöltődjön. Ezután a JIT fordító az alkalmazás belépései pontjához tartozó metódust natív gépi kódú utasításokká fordítja, majd átadja a vezérlést a lefordított natív utasításoknak. A fordítás, végrehajtás folyamatot a 2. ábra szemlélteti.



2. ábra Alkalmazás fordítása, végrehajtása

1.2.3. A .NET Framework fejlődése

A .NET Framework folyamatosan fejlődik, újabb és újabb szolgáltatásokkal, komponensekkel támogatja a fejlesztők munkáját, és teszi lehetővé, hogy többfélélőre eszközön, különböző környezetekben működjene az alkalmazások. A különböző bővítéseket új verziók formájában adták ki. Erről ad áttekintést a 3. ábra. Nagy változást hozott a 2006-as kiadású **.NET Framework 3.0** változat, amelyben új technológiák jelentek meg.

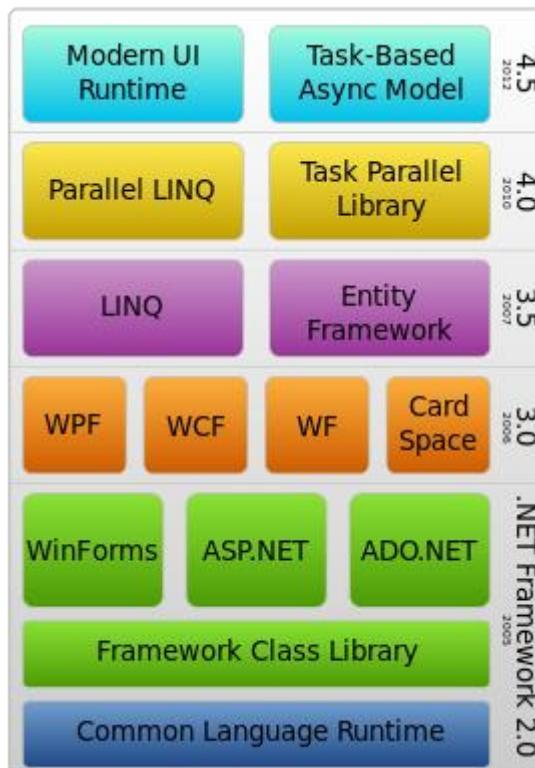
A **Windows Presentation Foundation (WPF)** egy vektorgrafikus rendszer, amely a klasszikus Windows Form alkalmazásoknál sokkal szebb grafikus megjelenítést, animációk készítését, videók lejátszását biztosító kódok írását teszi lehetővé. Leglényegesebb

tulajdonsága, hogy a megjelenítési felület kódja és az alkalmazás funkcionálitását biztosító kód teljesen elkülönül egymástól. A felhasználói felület kódja az Extensible Application Markup Language (XAML) nyelven készül. A felület és az üzleti logika szétválasztása lehetővé teszi, hogy a programozók és a látványos felületet készítő munkatársak könnyebben együtt tudjanak működni.

A **Windows Communication Foundation (WCF)** kommunikációs technológia olyan elosztott, szolgáltatásorientált alkalmazások készítésére alkalmas, amik egymáshoz kapcsolódva képesek működni. A WCF támogatást nyújt a webszolgáltatásokkal való kommunikációra, interoperábilis egyéb, SOAP protokollt támogató technológiával. A WCF kommunikációs réteg a .NET keretrendszer futtató platformjára (CLR) implementált osztályok halmaza. Ezek az osztályok a System.ServiceModel névterben találhatók.

A **WorkFlow Foundation (WF)** keretrendszer segítségével magas szintű, deklaratív nyelven valósíthatjuk meg az alkalmazások üzleti logikáját. Használatával tetszőleges jellegű üzleti logika implementálható. A **WF** szolgáltatásai kiemelten támogatják a hosszú lefutású, interaktív munkafolyamatok készítését.

A **.NET Framework 3.5** verzióban, amely 2007-ben jelent meg, további újdonságok épültek be, a **LINQ** és az **Entity Framework**. Ezek az új technológiák az adatelérést, adatmanipulációt hivatottak megkönnyíteni.



3. ábra A .NET Framework fejlődése

A **LINQ** (*Language Integrated Query*) komponens egy nyelvbe ágyazott lekérdező nyelv, melynek segítségével gyűjteményekből, adatbázisokból származó adatokat kérdezhetünk le, gyűjthetünk, módosíthatunk. Lekérdező kifejezések segítségével egyszerű vagy összetett szűréseket, csoportosításokat végző műveleteket adhatunk meg. A lekérdezés független az adatforrástól, ami lehet egy SQL adatbázis, egy XML dokumentum vagy egy gyűjtemény (collection), például egy egyszerű tömb vagy lista.

Az **Entity Framework** a koncepcionális adatmodell és a fizikai adatbázis közötti leképezést valósítja meg. Megfogalmazhatjuk az adatmodellt koncepcionális szinten, de meglévő adatbázisból is kiindulhatunk, vagyis a fizikai modellből. A két modell között az Entity Framework megteremti a kapcsolatot és elvégzi a leképezéssel kapcsolatos feladatokat. A modellalkotást grafikus felülettel támogatja. Többféle megközelítésmódban készíthetjük az alkalmazásainkat, **Code First**, **Database First** és **Design First**. A Code First megközelítés azt jelenti, hogy a fejlesztő először elkészíti az alkalmazását egy adott programozási nyelven, és az Entity Framework létrehozza az adatmodellt, amelyet leképez fizikai szintre is. A Database First metodika alkalmazásakor egy létező adatbázisból indulunk ki, amelynek entitásaiból az Entity Framework osztályokat generál. A Design First szemlélet alkalmazásakor először a fejlesztő megalkotja a koncepcionális adatmodellt, amelyből az Entity Framework a leképezés során elkészíti az adatmodellt az adatbázisszerveren.

A **.NET Framework 4.0** verziója 2010-ben jelent meg, amely új komponensek, a **PLINQ** és a **Parallel Task Library** bevezetésével lehetővé teszi, hogy a többmagos processzorral rendelkező számítógépeken teljesítménynövekedést érjünk el.

A **PLINQ** (Parallel LINQ), párhuzamos rendszerekben támogatja a LINQ-s lekérdezéseket. A mai többmagos, többprocesszoros rendszerekben megpróbálja teljesen kihasználni a rendelkezésre álló processzorokat. Ezt úgy éri el, hogy az adatlekérdezés során megpróbálja az adatforrást szegmensekre, részekre bontani, és ezeken a részeken egymással párhuzamosan történik a műveletek végrehajtása.

A **Framework 4.5** változata 2012-ben került a piacra. Legfontosabb újdonsága a megváltozott modern felhasználói felület és az aszinkron működés hatékonyabb támogatása. A tömbök tárolását tekintve újdonság, hogy a 64 bites platformokon 2 gigabyte-nál nagyobb méretű tömböket is használhatunk. Növelték a JIT fordító és a Garbage kollektor hatékonyságát is, 64 bites platformokon. Új programozási interfészt alakítottak ki a http alkalmazások számára. Ezekben túl sok egyéb újítás is bekerült az új .NET Framework verzióba.

1.2.4. A Common Type System

A CTS a típusok nyelvfüggetlen leírását tartalmazza az ECMA szabványnak megfelelően. A .NET keretrendszer által definiált típusok a *System* névvel kezdődő névterekben találhatók. minden típus, az interfésekkel kivéve, a *System.Object* típusból származik. Ebből következik, hogy a *System.Object* minden más típussal kompatibilis. Alapvetően két csoportba sorolhatjuk a típusokat, **érték-** és **referenciatípus**, a memoriában való elhelyezkedésüknek megfelelően. Az **értéktípusokat** a **veremben** (**stack**) vagy a kezelt **halomban** (**heap**), míg a **referenciatípusokat** minden esetben a **halomban** tároljuk.

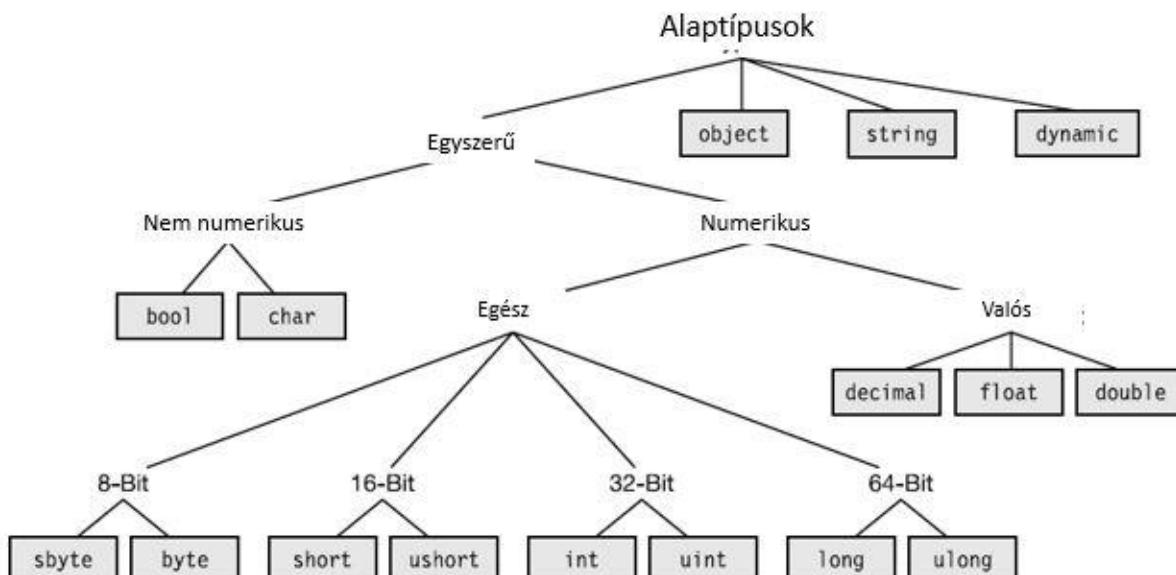
Az értéktípus (value type) jellemzője, hogy közvetlenül az értéket tárolja bitsorozat formájában a számára fenntartott memóriahelyen. Ilyen egyszerű értéktípus például a **System.Int32**, 32 bites egész, **System.Single**, 32 bites valós, **System.Boolean**, 8 bites logikai típus, vagyis az alaptípusok, a struktúrák (struct) és a felsorolt (enum) típus.

A referenciátípus (reference type) egy hivatkozást tartalmaz arra a memóriarészre, ahol az érték van. Maga az érték (bitsorozat) a felügyelt blokkban, az ún. managed heapben helyezkedik el. A legjellemzőbb példa a referenciátípusra az összes osztály, valamint a **System.String** sztring típus.

1.3. A C# programozási nyelv (*Szendrői E.*)

A C# programozási nyelv a .NET Framework fő programozási nyelve. A nyelvet fejlesztő csoport vezetője Anders Hejlsberg dán szoftvermérnök, aki korábban a Delphi és a Turbo Pascal nyelvek vezető fejlesztőjeként vált ismertté. Hasonlóan a keretrendszerhez, a C# nyelv is folyamatosan fejlődik, minden újabb verziója jelentős újításokat tartalmaz.

A C# nyelv objektumorientált, erősen típusos programozási nyelv, ami azt jelenti, hogy a programkódban szereplő összes változó és objektumpéldánynak jól definiált a típusa. A 4. ábrán a C# nyelv alaptípusai láthatók.



4. ábra A C# alaptípusai

A C# nyelvben minden, még a legalapvetőbb típusok is objektumok. A következő táblázatban az alaptípusokat (beépített típusokat) soroljuk fel, feltüntetve a System névtér azon típusát is, amelyre az adott típust leképezzük.

1. táblázat A C# beépített típusai

C# típus	Méret bitekben	.NET típus
sbyte	8 bites, előjeles egész szám	System.SByte
short	16 bites, előjeles egész szám	System.Int16
int	32 bites, előjeles egész szám	System.Int32
long	64 bites, előjeles egész szám	System.Int64
byte	8 bites, előjel nélküli egész szám	System.Byte
ushort	16 bites, előjel nélküli egész szám	System.UInt16
uint	32 bites, előjel nélküli egész szám	System.UInt32
ulong	64 bites, előjel nélküli egész szám	System.UInt64
char	16 bites, Unicode karakter	System.Char
bool	8 bites, logikai típus	System.Boolean
float	32 bites lebegőpontos szám	System.Single
double	64 bites, lebegőpontos szám	System.Double
decimal	128 bites fix pontosságú szám	System.Decimal
string	referenciatípus, Unicode karakterszorozat	System.String
object	minden előre definiált alap és felhasználó által létrehozott típus a System.Object-ból származik	System.Object

Annak, hogy minden alaptípus a System.Objectból származik (később még lesz róla szó), az a következménye, hogy minden típus örökli az object metódusait, többek között a ToString() metódust is, amely a változók értékének kiíratásakor játszik fontos szerepet.

A 16 előre definiált típuson túl a felhasználók saját típusokat hozhatnak létre. Ezek a következők:

- osztály (class)
- struktúra (struct)
- tömb (array)
- felsorolt (enum) típus
- interfész (interface)
- delegált (delegate).

A felsoroltak közül a struktúra (struct) és a felsorolt (enum) típus értéktípusok, míg a többi referenciatípus.

1.3.1. A C# nyelv szintaktikája

A C# nyelv utasításait pontosvesszővel (;) zárjuk le. A kis- és nagybetűk között a nyelv különbséget tesz, tehát a *pont* és a *Pont* a fordítóprogram számára nem ugyanaz. A programegységeket blokkokba foglaljuk, kapcsos zárójelek { és } használatával.

Minden programnyelvnek vannak kulcsszavai, amelyeket a saját jelentésükön kívül nem használhatunk másra, mert a fordító nem tudná értelmezni. A C# nyelvnek 77 kulcsszava van, melyeket a 2. táblázatban gyűjtöttünk egybe.

2. táblázat C# kulcsszavak

abstract	const	extern	int	out	short	Typeof
as	continue	false	interface	override	Sizeof	Uint
base	decimal	finally	internal	params	Stackalloc	Ulong
bool	default	fixed	is	private	Static	unchecked
break	delegate	float	lock	protected	String	unsafe
byte	do	for	long	public	Struct	ushort
case	double	foreach	namespace	readonly	Switch	using
catch	else	goto	new	ref	This	virtual
char	enum	if	null	return	Throw	volatile
checked	event	implicit	object	sbyte	True	void
class	explicit	In	operator	sealed	Try	while

Változók

A változó egy név, amellyel egy memóriaterületre hivatkozunk. A változónév első karaktere csak betű vagy alulvonás (_) karakter lehet. A többi karakter lehet szám is. A C# nyelvben konvenció szerint a változónevek mindig kisbetűvel kezdődnek. Változó létrehozása a C# nyelvben a változó típusának és nevének megadásával történik. A változó létrehozásakor lehetőség van arra, hogy kezdőértéket rendeljünk hozzá. Például így:

```
int oldal=7;  
double hossz=120.5;
```

Ha nem adunk meg explicit módon kezdőértéket, akkor a fordító a változó típusának megfelelő 0 értékkel tölti fel a változó által lefoglalt memóriaterületet. Referenciátípusok esetén az alapértelmezett érték *null* érték.

Implicit típusú változók

Mivel a C# egy erősen típusos nyelv, minden deklarált változónak rendelkeznie kell egy adott típussal. A változó tartalmának tárolásakor egy típus is hozzárendelődik a lefoglalt tárterülethez. Összetett típusoknál (pl. LINQ lekérdezés eredményét tartalmazó változó) nehézkes a típust megadni, helyette implicit típusú változót deklarálunk a **var** kulcsszóval.

Ha a **var** kulcsszóval deklarálunk egy lokális változót, azzal azt kérjük a fordítótól, hogy egy lokális memóriahelyet foglaljon le és rendeljen hozzá egy feltételezett típust. Fordításkor, a változó inicializálásakor már rendelkezésre fog állni annyi információ, hogy a fordító kikövetkeztesse a változó konkrét típusát, anélkül, hogy mi ezt explicit módon megadnánk.

Például:

```
var i=10; // a változó típusát a fordító ki tudja következtetni az  
// értékkadás során kapott értékből
```

Konstansok

Gyakran van szükségünk olyan értékekre, amelyeket a program futása során nem szabad megváltoztatnunk. Az ilyen típusú értékeket konstansként definiáljuk.

```
const int gyartev = 2012; // később az érték nem változtatható meg  
gyartev=2015 // hibát jelez a fordító
```

A konstansnak egyetlen egyszer adhatunk értéket, mégpedig kötelezően a deklarációtól. Bármilyen későbbi értékmódosítás fordítási hibát okoz.

Operátorok

A programokban a változókkal, a bennük tárolt adatokkal műveleteket végzünk. A programok végrehajtandó utasításokat tartalmaznak. Az utasításokban különböző algebrai, relációs és logikai műveletek vagy ezek kombinációit tartalmazó kifejezések vannak. A kifejezésekben szereplő változókat, konstansokat együttesen *operandusoknak*, a műveleteket *operátoroknak* nevezzük.

Az alapvető **aritmetikai** műveleteket numerikus adatokon végezhetjük el. A műveletek felsorolása a 3. táblázatban látható.

3. táblázat Aritmetikai operátorok

Művelet	Operátor
Összeadás	+
Kivonás	-
Szorzás	*
Osztás	/
Maradékképzés (modulus)	%

A maradékképzés műveletét a programozási nyelvek többségében csak egész típusú változókon végezhetjük el. A C# nyelv megengedi, hogy lebegőpontos változók vegyenek részt a műveletben.

```
int szam1=65, szam2=3;  
double szam3 = 65.5, szam4 = 3.1;  
Console.WriteLine("egész osztás maradéka: {0}", szam1 % szam2);  
Console.WriteLine("lebegőpontos osztás maradéka {0}", szam3 % szam4);
```

Érdekes eredményeket kapunk, ha negatív értékekkel végezzük el a műveletet. Álljon itt néhány példa: $-3 \% 5 = -3$; $5 \% -3 = 2$; $-5 \% -3 = -2$; Az eredmény előjele az osztandó előjelével lesz azonos.

Inkrementáló, dekrementáló operátorok

A programírás során gyakran előfordul, hogy egy változó értékét eggyel kívánjuk növelni vagy csökkenteni. Ezt könnyen elvégezhetjük az inkrementáló vagy dekrementáló operátorokkal. A műveletet a `++` vagy a `-` szimbólum jelzi. Nem mindegy, hogy a szimbólum a változó neve előtt (prefix) vagy a változó neve után (postfix) jelenik meg. Amennyiben a változó neve előtt szerepel a `++` vagy `-` operátor, akkor a változóban tárolt értéket növeli eggyel vagy csökkenti eggyel és az így kapott értéket használjuk fel. Ha a változó neve után szerepel a `++` vagy `-` operátor, akkor előbb felhasználjuk a változóban tárolt értéket, majd ezt követően növeli vagy csökkenti a változó értékét a futató környezet. Az alábbi kiíratási utasításban az `x` értékét berakjuk a kiírandó stringbe, majd megnöveljük `x` értékét eggyel (101 lesz az értéke), elhelyezünk egy szóközt a stringbe, majd hozzáfűzzük a stringhez `x` eggyel növelt (102) értékét, s ezután a teljes stringet kiírjuk.

```
int x = 100;
Console.WriteLine(x++ + " " + ++x); //100-at majd 102-t ír a konzolra
- - - - -
```

Másik példa:

```
int szamlal = 0, eredm = 0, szam1 = 10;
szamlal++;
erdem = szamlal++ * --szam1 + 100;
Console.WriteLine("eredmény: {0}", eredm); // az eredmény 109 lesz
```

A változók értékének inicializálása után a `szamlal` változó értéke növekszik eggyel, értéke 1 lesz. Ezután az értékadó utasításra adódik a vezérlés. Az értékadó utasítás végrehajtása az értékadásjel jobb oldalán szereplő kifejezés kiértékelésével kezdődik. A gép kiolvassa a `szamlal` értékét, ami jelenleg 1, ezután növeli eggyel az értéket (postfix inkrementálás, `szamlal` új értéke 2), majd csökkenti a `szam1` változóban tárolt értéket eggyel, ez 9 lesz, majd elvégzi a szorzás műveletet, azaz $1 \cdot 9 = 9$. Ezután az összeadásra kerül sor, a szorzat eredményéhez (9) hozzáad 100-at. Így kapjuk a kiíratásban a 109 végeredményt.

Relációs operátorok

A relációs operátorok segítségével értékeket hasonlíthatunk össze. A művelet eredménye **logikai true** vagy **false** érték lesz.

A relációs kifejezésekben a matematikában megszokott relációs operátorokat használhatjuk összehasonlításra, melyek a következők: `>`; `>=`; `<`; `<=`; `==`; `!=`. Ügyelni kell arra, hogy az egyenlőségi reláció két egyenlőségi jelből áll, a nem egyenlőségi reláció a `!=`-jel és az `=` jel együtt.

Logikai operátorok

A következő logikai műveletekkel logikai kifejezések adhatók meg: ÉS: `&&`, VAGY: `||`, TAGADÁS (negáció) `!()`. A negáció kivételével minden egyik művelet kétoperandusú művelet.

A kifejezés kiértékelése balról jobbra történik. Ha a logikai ÉS műveleti jel előtt szereplő logikai érték hamis, akkor tovább már nem folytatódik a kifejezés kiértékelése, hiszen mindenképpen hamis eredményt kapunk. Hasonlóan, a logikai VAGY művelet esetén, ha a

műveleti jel előtti logikai érték igaz, akkor nem folytatódik a kifejezés kiértékelése, mert az eredmény mindenképpen igaz lesz. Ezt hívák lusta kiértékelésnek.

A logikai tagadás egyoperandusú művelet. Ha az operandus értéke hamis, a művelet eredménye igaz lesz. Ha az operandus értéke igaz, akkor a művelet eredménye hamis lesz.

Feltételes operátor

Háromoperandusú operátor. Segítségével egyszerű döntési (if) utasításokat helyettesíthetünk. Szintaxisa: **feltétel ? igaz-ág: hamis-ág;**

Először a ? jel előtti feltétel kerül kiértékelésre, majd a gép megvizsgálja, hogy az eredmény igaz-e. Ha igaz, akkor a ? jel utáni „igaz-ág”-at hajtja végre. Ha a feltétel kiértékelésének eredménye hamis lesz, akkor a : utáni, „hamis-ág”-at. Használatát az alábbi egyszerű példával szemléltetjük.

```
int n;  
Console.WriteLine("Kérek egy egész számot: ");  
n = Convert.ToInt32(Console.ReadLine());  
Console.WriteLine(n % 2 == 0 ? "A szám páros szám" : "A szám páratlan szám");
```

Értékadó operátor

Leggyakrabban használt művelet, segítségével egy változónak adhatunk értéket. Végrehajtása során az értékadásjel bal oldalán szereplő változó megkapja az értékadásjel jobb oldalán álló konstans vagy változó vagy kifejezés értékét. Az értékadás jele az egyenlőségjel: =. Az értékadás során ügyelni kell arra, hogy egy változó csak a típusának megfelelő értéket vehet fel, tehát az értékadás jobb oldalán szereplő konstans, kifejezés vagy változó típusának meg kell egyeznie a bal oldalon álló változó típusával, vagy megfelelő típuskonverzió alkalmazásával átalakíthatónak kell lennie.

Műveletek végrehajtási sorrendje

Az utasítások gyakran összetett kifejezéseket tartalmaznak, amelyekben aritmetikai, relációs és akár logikai műveletek is szerepelnek. A fordítónak fel kell állítania egy sorrendet a műveletek között, amely alapján kiértékeli a kifejezést. Általános szabály, hogy balról jobbra haladva végzi a műveletek végrehajtását egy úgynvezett precedencia szabálynak megfelelően. Ettől a sorrendtől zárójelek alkalmazásával tudjuk eltéríteni. A műveleti precedencia a következő (a kisebb sorszámról műveletet kiértékelni előbb):

1. zárójel, postfix inkrementáló és dekrementáló operátor
2. előjel, logikai tagadás, prefix inkrementáló és dekrementáló operátor
3. szorzás, osztás, maradékképzés (modulus)
4. összeadás, kivonás
5. relációs műveletek ($<$, \leq , $>$, \geq)
6. egyenlő, nem egyenlő operátorok
7. logikai ÉS művelet
8. logikai VAGY művelet
9. feltételes operátor.

Vezérlő szerkezetek

Szelekció

A számítógép a fizikai sorrendjüknek megfelelően hajtja végre az utasításokat. Ettől a szekvenciális végrehajtási sorrendtől vezérlő szerkezetek alkalmazásával téríthatjuk el. Gyakran egy feltétel kiértékelésének eredményétől függ, hogy mely utasításokat hajtjuk végre. Az ilyen döntéseket a programokban **if-else** szerkezet alkalmazásával valósítjuk meg. A döntési szerkezet lehet **egyágú**, **kétágú** vagy **többágú**. A többágú döntéseket egy speciális szerkezettel, a **switch-case** utasítással oldjuk meg.

Egyágú szelekció esetén, csak akkor hajtunk végre bizonyos utasításokat, ha a vizsgált feltétel igaznak bizonyul. Szintaxisa:

```
if (feltetel)
{
    utasitas;
}
```

Kétágú szelekció esetén mind az igaz-ágon, mind a hamis-ágon szerepelnek utasítások, s a vizsgált feltétel kiértékelésének eredményétől függ, hogy melyik ág fut le. Vagy csak az igaz-ág utasításai, vagy csak a hamis ág utasításai lesznek végrehajtva. A kétágú szelekció szintaxisa:

```
if (feltetel)
{
    utasitas;
}
else
{
    utasitas2;
}
```

Arra is lehetőség van, hogy további feltételeket is megvizsgálunk, ilyenkor **if - else if - else** szerkezetet használunk. A döntési szerkezetek egymásba ágyazhatók.

A **switch-case** szerkezetet akkor használjuk, ha egy változó sok értéket vehet fel és azt szeretnénk eldönteni, hogy éppen melyik értéket tartalmazza. A case utasításrészben adjuk meg, hogy mi történjék, ha a vizsgált változó éppen a case utasításban megadott értéket veszi fel. A vizsgált érték egész, string és felsorolt (enum) típusú lehet. A C# nyelvben a case ágakat kötelezően break utasítással kell lezárni. Ez csak abban az esetben hagyható el, ha az adott ág nem tartalmaz semmilyen utasítást.

A következő példa a switch-case szerkezet használatát mutatja be:

```

switch (kockadobas)
{
    case 1:
    case 3:
    case 5:
        Console.WriteLine("páratlant dobtak");
        break;
    case 2:
    case 4:
    case 6:
        Console.WriteLine("párost dobtak");
        break;
    default:
        Console.WriteLine("nem 1_6 közötti értéket dobtak");
        break;
}

```

A default ágra akkor adódik a vezérlés, ha a vizsgált változó értéke egyik case utasításban felsorolt értékkel sem egyezik meg.

Ciklusok

Feltételes ciklusok: Valamilyen feltételtől függően ismétlünk meg egy tevékenységet vagy tevékenységsorozatot. Két típusát különböztetjük meg, az *elöl tesztelő* és a *hátul tesztelő* ciklust.

```

while (feltétel)
{
    utasítások;
}

```

Az **elöl tesztelő**, más néven **while** ciklus esetén, a feltétel kiértékelése a ciklusmag végrehajtása előtt történik. Lehet, hogy egyszer sem fut le a ciklus, amennyiben a feltétel már az első vizsgálatkor hamis. Mindaddig, amíg a feltétel igaz, a ciklus lefut, azaz a ciklusmag tevékenységei végrehajtódnak. Ha a feltétel hamissá válik, a ciklus futása befejeződik, és a ciklus utáni tevékenységgel folytatódik az algoritmus. A programozó felelőssége, hogy gondoskodjon arról, hogy a ciklus végrehajtási feltétele valamikor hamissá váljék, lehetővé téve a ciklus befejezését.

A **hátul tesztelő**, más néven **do-while** ciklus esetén, a feltétel kiértékelése a ciklusmag végrehajtása után történik. A ciklus egyszer mindenképpen lefut. Amennyiben a feltétel hamis, a ciklus futása befejeződik, és a ciklus utáni tevékenység végrehajtásával folytatódik az algoritmus. Ha a feltétel igaz, a ciklus fut tovább. Itt is a programozónak kell arról gondoskodnia, hogy a végrehajtási feltétel hamissá váljon és befejeződjön a ciklus végrehajtása.

```

do
{
    utasítások;
} while (feltétel);

```

A **léptetéses** vagy **for** ciklus az elöl tesztelő ciklus speciális esete. Gyakran előfordul, hogy egy tevékenységsorozatot megadott számú alkalommal kell végrehajtani, azaz ismert a cikluslefutások száma. Ekkor érdemes a for ciklust használni. A cikluslefutások száma a

ciklusváltozó kezdő- és végértéke, valamint a lépésköz (növekmény) ismeretében kiszámítható a követő formula alkalmazásával: [(végérték – kezdőérték) / lépésköz]+ 1.

```
for (int i = 0; i < n; i++)
{
    sum += i;
}
```

Amennyiben a ciklusváltozó kezdőértéke nagyobb, mint a végértéke és a lépésköz pozitív, akkor a ciklusmag egyszer sem fog lefutni. Hasonlóan, ha a lépésköz negatív és a ciklusváltozó kezdőértéke kisebb, mint a végértéke, akkor a ciklustörzs egyszer sem lesz végrehajtva. Egy ciklusban több azonos típusú ciklusváltozó is lehet. **For** ciklus blokkjában nem lehet deklarálni a ciklusváltozóval azonos nevű változót.

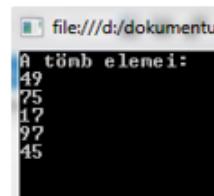
A **foreach** ciklust tömbök vagy gyűjtemények bejárására használjuk. A foreach utasítás törzsében tilos az iterációs változót módosítani, ezt kizárolag csak olvasható változóként kezelhetjük.

Autasítás általános alakja:

```
foreach (tipus elemnév in adatlista) {
    utasítások; }
```

A ciklusfejben deklarálunk egy változót, a példában *elemnév*, majd az **in** kulcsszó következik, amivel megadjuk, hogy melyik gyűjteményen haladunk végig. Az iterációs változó (a példában *elemnév*) felveszi a gyűjtemény aktuális elemének értékét. Ezért fontos, hogy az iterációs változó nem kaphat értéket, típusa pedig megegyezik a gyűjtemény elemeinek típusával. A konkrét típusmegadást helyettesíthetjük a **var** kulcsszóval, ezzel a fordítóra bízzuk, hogy a gyűjtemény típusa alapján döntse el az iterációs változó típusát. A ciklusfejben az (**int** elem **in** vektor) helyett írhattuk volna a (**var** elem **in** vektor) kifejezést is.

```
// kiíratjuk a vektor elemeit foreach ciklussal
Console.WriteLine("A tömb elemei:");
foreach (int elem in vektor)
{
    Console.WriteLine(elem);
}
Console.ReadKey();
```



Metódusok

A metódus utasítások logikailag összefüggő csoportja, melynek önálló neve és visszatérési értéke van. (Más programozási nyelven függvény, eljárás, szubrutin). Egy metódus segítségével megváltoztathatjuk egy objektum állapotát vagy kiolvashatjuk tulajdonságainak értékét, leírhatjuk viselkedését. Egy metódust meghívhatunk a program bármelyik pontjából vagy akár egy másik programból. Eltéren a C és C++ nyelvktől, a C# nyelvben nincsenek

globális metódusok, a metódusok csak osztályon belül léteznek, csak osztályon belül definiálhatók.

A metódus definiálásakor meg kell adni a metódusfejet és a metódustörzset. A metódusfej tartalmazza az úgynevezett hozzáférési módosítót, a visszatérési értéket, a metódus nevét és a paraméterlistát. A metódustörzs a lokális változók deklarációját és a metódusban végrehajtandó utasításokat tartalmazza, valamint a *return* utasítást a visszatérési értékkel együtt, ha van visszatérési érték. A metódus definíció a következő formájú:

```
[módosító] visszatérési érték típusa MetódusNév ([paraméterlista])
{
    // metódus törzse
    lokális deklarációk;
    utasítások;
    [ return [kifejezés] ; ]
}
```

A *metódus* neve konvenció szerint minden nagybetűvel kezdődik. A módosító, az úgynevezett *hozzáférés* (*láthatóság*) megadására szolgál. Ha nem adjuk meg, akkor a hozzáférés az alapértelmezett, *private* lesz.

4. táblázat C# hozzáférési módosítók

Módosító	Magyarázat
public	nincs korlátozás, bárhonnet elérhető
protected	csak abból az osztályból, ahol létrehozták, vagy annak leszármazottjából látható, érhető el
internal	az aktuális projektből látható, érhető el
protected internal	az aktuális projektből vagy a létrehozó osztály leszármazottjából érhető el
private	csak az osztályon belül látható, érhető el

A *paraméterlista* arra szolgál, hogy a hívó programmodul értékeit adjon át a meghívott metódusnak, amellyel az el tudja végezni a feladatát. A paraméterlista elemeit gömbölyű zárójelbe kell tenni és a típusukat meg kell adni. A paraméterlista lehet üres is, ha nem szükséges kívülről adatokat átadni a metódus működéséhez. A *return* utasítás hatására visszaadódik a vezérlés a hívó programrészbe. Ha a metódusnak van visszatérési típusa, akkor legalább egy *return* utasításnak szerepelnie kell a metódus törzsében. A metódus (visszatérési érték) típusa a következő lehet:

- Értéktípus, felsorolt típus, struktúra
- Referenciátípus (tömb, osztály, interfész, delegált)
- Ha nincs visszatérési érték, akkor **void**.

Ha a metódus rendelkezik visszatérési típussal, akkor függvénynek is szoktuk nevezni. Hívása értékkedás jobb oldalán, kifejezésben vagy más metódus paramétereként lehetséges.

A **void** típusú metódust más nyelveken eljárásnak nevezzük. Hívása önálló utasításként jelenik meg a programban, pl. *Kiír(a,b)*;

A metódus nevét és paraméterlistáját együttesen **szignatúrának** (aláírásnak) nevezzük. Egy osztályban több azonos nevű metódus definiálható, ha a paraméterlistájuk különböző. A metódus deklarációjában szereplő paraméterlistát szokás *formális paramétereknek* is nevezni. A metódus *hívásakor* megadott argumentumok alkotják az *aktuális paramétereket*. Az aktuális paraméterlistának típusban, sorrendben és számban meg kell egyeznie a formális paraméterlistával. Az aktuális paraméterlistán nemcsak változót, hanem konstanst vagy kifejezést is megfeleltethetünk egy formális paraméternek.

Paramétertípusok, paraméterátadás

A C# programozási nyelvben érték és cím szerinti (referencia) paraméterátadás lehetséges. Az érték szerinti paraméterátadás az alapértelmezés.

Érték szerinti paraméterátadáskor a metódusban létrejövő lokális paraméterváltozóba átmásolódik a híváskor megadott adat vagy hivatkozás. Az aktuális paraméterlista értékei inicializálják a formális paraméterlistában szereplő változókat. A hívó oldalon szereplő aktuális paraméterek és a metódus formális paraméterlistáján szereplő változók, más-más memóriaterületet foglalnak el. Példaként nézzük meg két változó tartalmának megcserélését végző metódus működését. A hívó modulban meghívjuk a **csere** metódust a **szam1** és **szam2** aktuális paraméterekkel.

```
csere(szam1, szam2);
          ↓         ↓
          a         b
A szam1 és szam2 értéke átadódik
a formális paramétereknek, a-nak
és b-nek

private static void csere(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    Console.WriteLine("A csere után a={0}, b={1}", a, b);
}
```

A hívást megelőzően és a hívást közvetlen követően kiírjuk a szam1 és szam2 értékét. A metódus is kiírja a lokális paraméterek értékét. Az eredmény:

```
A csere előtt szam1=3, szam2=5
A csere után a=5, b=3
A csere után szam1=3, szam2=5
```

Az eredményből jól látható, hogy a metódusban megtörtént a csere, de ez nem volt hatással a hívó oldal szam1 és szam2 változókban tárolt értékre, mivel azok elkülönült memóriaterületen vannak. Tehát a metódus nem tudta a hívó oldal változóinak értékét megváltoztatni.

A **cím szerinti (referencia) paraméterátadást** a **ref** kulcsszóval jelezzük. A metódus hívásakor a megadott objektumra mutató hivatkozás adódik át a formális paraméternek. A paraméteren keresztül magát az eredeti objektumot érjük el, legyen akár érték vagy hivatkozás típusú. Az előző példán, a számok cseréjén mutatjuk be a cím szerinti

paraméterátadást. A metódus formális paraméterlistáján és a hívásakor az aktuális paraméterlistán is a **ref** kulcsszóval jelezzük, hogy hivatkozást adunk át a metódusnak.

```
csere( ref szam1, ref szam2);  
+-----+  
| reference |  
private static void csere(ref int a, ref int b)  
{  
    int temp;  
    temp = a;  
    a = b;  
    b = temp;  
    Console.WriteLine("A csere után a={0}, b={1}", a, b);  
}
```

A szam1 és szam2 memóriacíme átadódik a formális paramétereknek, a-nak és b-nek, vagyis a ugyanarra a memória helyre mutat mint szam1 és b ugyanarra amire szam2

A futási eredmény:

```
A csere előtt szam1=3, szam2=5  
A csere után a=5, b=3  
A csere után szam1=5, szam2=3
```

Az eredményből jól látható, hogy a metódus, mivel hozzáfért a szam1 és szam2 változók memóriacíméhez, az ott tárolt értékekkel hajtotta végre a cserét. Tehát nemcsak a metódusban, hanem az öt hívó eljárásban is látszik, hogy megtörtént a csere.

A **kimenő (output) paramétereket** (**out** módosító) arra használjuk, hogy a metódusból adjunk vissza értékeket a hívó programkódba. Viselkedésük nagyon hasonló a referenciátipusú paraméterekhez. Az aktuális paraméterlistán kifejezés vagy konstans nem feleltethető meg az out módosítóval rendelkező formális paraméternek, csak változó. mindenéppen értéket kell kapnia a metódusból való kilépés előtt. Az **out** kulcsszóval jelzett formális és aktuális paraméter ugyanarra a memóriahelyre mutat, hasonlóan a referenciátipushoz. A metódus hívása előtt az aktuális paramétert nem kell inicializálni.

```
int szam1 = 3, szam2 = 6;  
float atlag;  
  
AtlagSzamitas(szam1, szam2, out atlag);  
Console.WriteLine("A szám1={0} és szám2={1 } átlaga {2:F2}", szam1, szam2, atlag);  
  
+-----+  
| reference |  
private static void AtlagSzamitas(int a, int b, out float atl)  
{  
    atl = (a + b) / 2.0F;  
}
```

Az **atlag** aktuális paraméter és az **atl** formális paraméter ugyanarra a memóriacímre mutat.

A metódusban, mivel két egész számot kell elosztanunk kettővel az átlag kiszámításához, hogy helyes eredményt kapjunk, típuskényszerítéssel az osztót **float** típusúvá alakítottuk.

A futás eredménye:

```
A szám1=3 és szám2=6 átlaga 4,50:
```

Paramétertömbök (**params** módosító) lehetővé teszi, hogy nulla vagy több aktuális paraméter tartozék a formális paraméterhez. Egy paraméterlistán csak egy paramétertömb lehet, s ennek kell az utolsónak lennie. A paramétertömb csak azonos típusú elemeket tartalmazhat.

```
int[] vekt = new int[n];
atlag=Atlag(vekt);
private static double Atlag(params int[] szamok){
```

Az 5. táblázatban a metódusok formális paraméterlistájában használható paramétertípusokat foglaltuk össze.

5. táblázat Metódus paramétertípusok

Paraméter típus	Módosító	Deklarációkor meg kell adni?	Híváskor meg kell adni?	Működése
Érték	nincs			A rendszer az aktuális paraméter értékét átmásolja a formális paraméterbe.
Referencia	ref	Igen	Igen	A formális paraméter és az aktuális paraméter ugyanarra a memóriacímre mutat.
Output	out	Igen	Igen	A paraméter csak visszatérő, kimenő értéket tartalmaz.
Paramétertömb	params	Igen	Nem	Lehetővé teszi, hogy változó számú aktuális paramétert adjunk át a metódusnak.

Egy másik példát is bemutatunk az érték szerinti, referencia szerinti paraméterátadásra és az out használatára. A példában egy henger térfogatát számoljuk ki. A példában bemutatjuk a metódusok túlterhelésének, **overloading** működését is. Egy osztályban lehet több azonos nevű metódust is létrehozni, ha a paraméterlistájuk különböző, vagyis a szignatúrájuk eltérő.

A példa kódja a következő:

```
using System;
static void Main(string[] args)
{
    /* Henger térfogatának számítása metódus alkalmazásával.
     * Példában különböző paraméterátadási lehetőségeket mutatunk be */
    float sugar, magassag, terfogat;
    Console.WriteLine("Kérlek adj meg a henger sugarát:");
    sugar = float.Parse(Console.ReadLine());
    Console.WriteLine("Kérlek adj meg a henger magasságát:");
    magassag = float.Parse(Console.ReadLine());
    // Érték szerinti paraméterátadás
    terfogat = HengerTerfogat(sugar, magassag);
    Console.WriteLine("A henger térfogata {0:F2}", terfogat);
    // Címszerinti paraméterátadás
    terfogat = HengerTerfogat(ref sugar, ref magassag);
    Console.WriteLine("A henger térfogata {0:F2}", terfogat);
    // Out paraméter használata
    HengerTerfogat(sugar, magassag, out terfogat);
    Console.WriteLine("A henger térfogata {0:F2}", terfogat);
    Console.ReadKey();
}
// Reference
static float HengerTerfogat(float sug, float mag) // Érték szerinti paraméterátadás
{
    return sug * sug * (float)Math.PI * mag;
}
// Reference
static float HengerTerfogat(ref float sug, ref float mag) // Ref, címszerinti paraméterátadás
{
    return sug * sug * (float)Math.PI * mag;
}
// Reference
static void HengerTerfogat(float sug, float mag, out float terf) // Out paraméter használata
{
    terf = sug * sug * (float)Math.PI * mag;
}
```

A futási eredmény:

```
Kérlek adj meg a henger sugarát:3,5
Kérlek adj meg a henger magasságát:6
A henger térfogata 230,91
A henger térfogata 230,91
A henger térfogata 230,91
```

Nevesített és alapértelmezett paraméterek

A C# 4.0 verzió egyik újítása a **nevesített paraméterek** használata. Ezzel a fejlesztők munkáját könnyítették meg. Ha nem emlékszünk pontosan a formális paraméterek sorrendjére, akkor az aktuális paraméterlistán úgy adhatjuk meg az argumentumot, hogy megadjuk az adott formális paraméter nevét, amelyet kettősponttal zárunk le, majd megadjuk az argumentum értékét. Ez lehetővé teszi azt is, hogy bármilyen sorrendben adjuk meg az aktuális paramétereket. Nem kell betartanunk azt a szabályt, hogy sorrendben is egyeznie kell a két paraméterlistának. A metódus definíciójában ez a lehetőség semmilyen változást nem jelent.

```

terulet = TeglalapTer(magassag: b, hossz: a);

private static double TeglalapTer(int hossz, int magassag)
{
    return hossz * magassag;
}

```

A másik újítás az **alapértelmezett paraméterek**, amelyek segítségével alapértelmezett értéket rendelhetünk hozzá egy formális paraméterhez. Ha az aktuális paraméterlistán nem kap értéket a paraméter, akkor az alapértelmezett értékkel dolgozik a metódus. A formális paraméterlista több alapértelmezett paramétert is tartalmazhat, azonban ezeknek minden a lista végén kell lenniük.

```

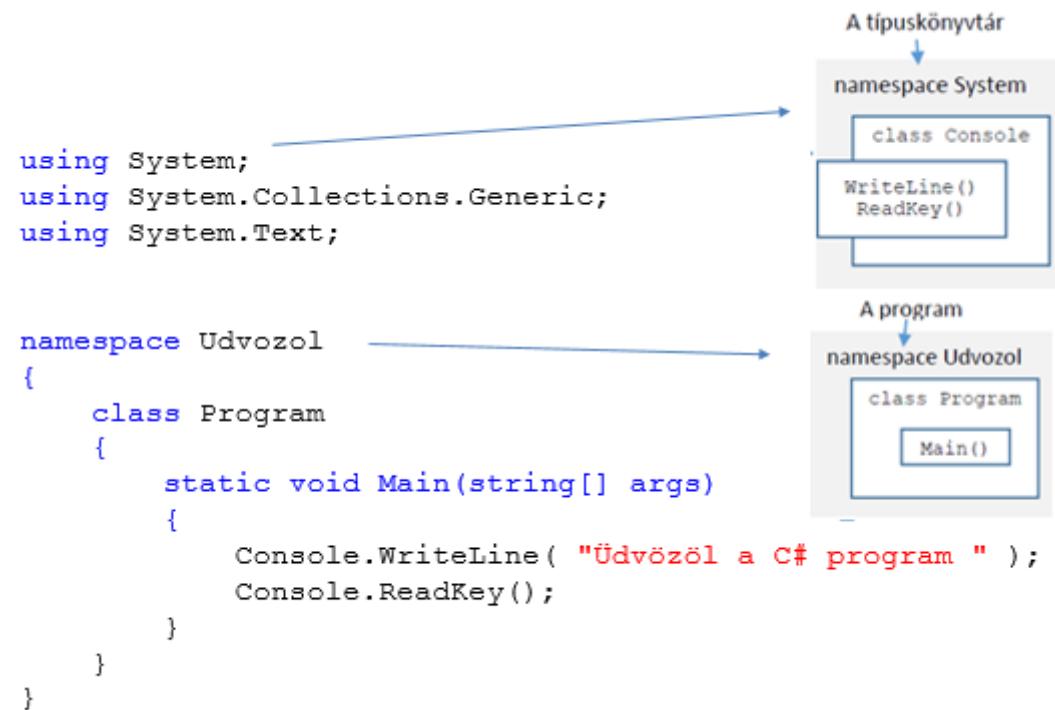
terulet = TeglalapTer(4, 7); // explicit érték a szélességnek
terulet = TeglalapTer(3);   // implicit szélesség érték

private static double TeglalapTer(int hossz, int szelesseg=6)
{
    return hossz * szelesseg;
}

```

1.3.2. A C# program szerkezete

A C# nyelv osztályok sokasága, amelyek együttműködnek egymással. A programot alkotó osztályok több állományban helyezkedhetnek el. Az 5. ábra a C# program szerkezetét mutatja:



5. ábra A C# program szerkezete

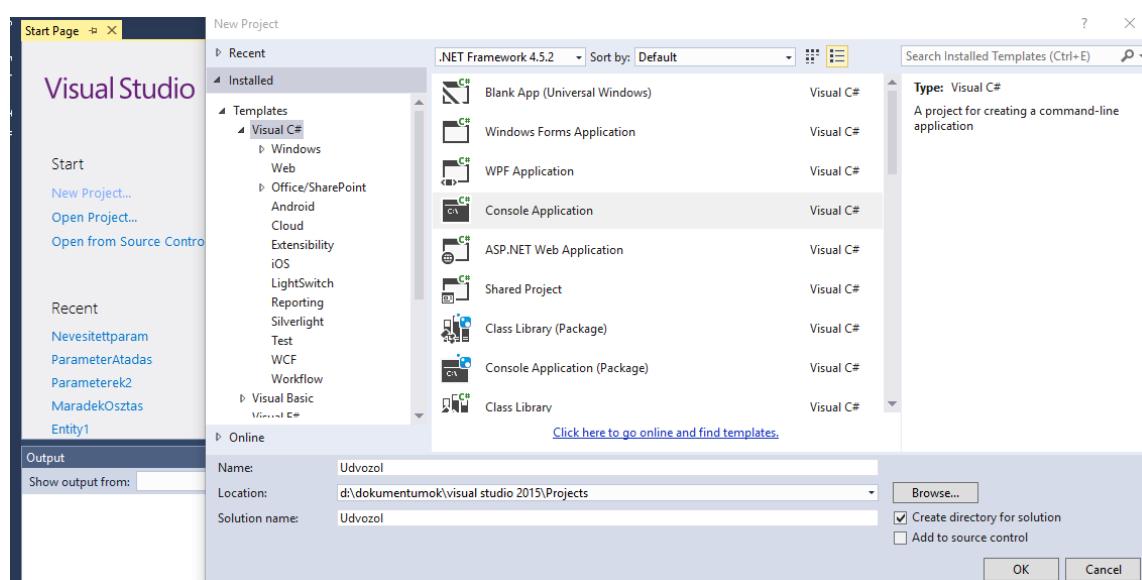
A program kódja a Program.cs szöveges fájlban van. Az alkalmazásunk számára létrejön egy *Udvozol* nevű névtér, amely név azonos az alkalmazás (projekt) nevével. Ebben a névtérben keletkezik automatikusan egy Program nevű osztály, benne a statikus Main() metódussal. A Main() metódus a program belépési pontja, ezzel kezdődik a program végrehajtása. Ebbe a metódusba írhatjuk az utasításainkat. A Main() metódus ebben az egyszerű programban mindössze két utasítássort tartalmaz.

A Main metódusnak nincs visszatérési értéke, ezt a *void* kulcsszó jelzi. A *static* módosító azt jelenti, hogy objektumpéldány létrehozása nélkül a metódus meghívható. A Main metódusnak egyetlen paramtere van, egy string tömb. Ezt parancssori paraméter megadására használhatjuk. Windows Form alkalmazások esetén a Main hozza létre a felhasználói felületet és az eseményvezérelt futtatást biztosító háttérkódot.

A programfájl elején látható **using** direktívák a System névtérre és annak néhány alterére hivatkoznak, és automatikusan bekerülnek a kódba, amikor egy új alkalmazást hozunk létre a Visual Studióban. A using direktívák a fordító számára megmondják, mely névterekben keresse a programkódban hivatkozott osztályokat, azok metódusait. Az ábrán látható, hogy a System névtérben lévő *Console* osztály szolgáltatásait vesszük majd igénybe, amely többek között az alapértelmezett beviteli-kiviteli eszközökkel (billentyűzet és képernyő) való kommunikációt támogató metódusokat tartalmaz. Az osztály *WriteLine()*, valamint a *ReadKey()* metódusát hívtuk meg a programból.

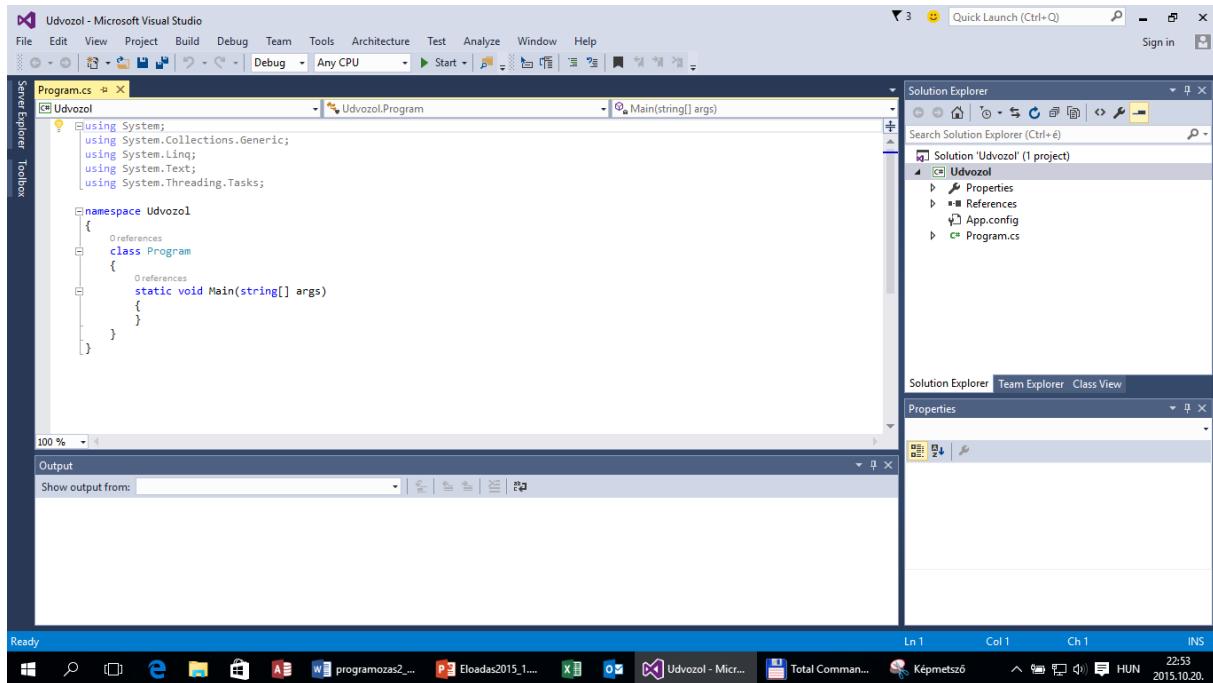
A *WriteLine()* metódussal a képernyőre írunk egy karaktersorozatot, majd a kurzor a következő sor elejére áll. A *ReadKey()* metódus egy billentyűleütésre vár. Erre a metódusra azért volt szükség, hogy legyen időnk megtekinteni a kiíratás eredményét, mielőtt a programfutás ablaka becsukódna. Ugyanis, ha konzolalkalmazást hozunk létre, az a Command ablakban fut, amely azonnal becsukódik, amint a program futása véget ér.

A Visual Studio fejlesztő környezet



6. ábra A Visual Studio

A Visual Studio indító ablakában tudunk új alkalmazást létrehozni, a **New Project** menü kiválasztásával. A felbukkanó ablakban tudjuk kiválasztani, hogy milyen nyelven és milyen típusú alkalmazást szeretnénk készíteni. Az alkalmazás nevének és helyének megadása után jutunk el a munkafelületre.



7. ábra A Visual Studio munkaablakai

A munkafelület 4 ablakból áll. Az első ablakban magát a programot szerkeszthetjük. Tőle jobbra van az úgynevezett Solution Explorer ablak. Ebben az ablakban látható az alkalmazásunk mappastruktúrája és azoknak a fájloknak az összessége, amelyek az alkalmazáshoz tartoznak. Itt látható a Program.cs fájl is. Ebben az ablakban válthatunk a Team Explorer és a Class View nézetre is, ha szükséges.

A Solution Explorer ablak alatt van a Properties ablak, ahol megtekinthetjük az alkalmazáshoz kapcsolódó tulajdonságokat. A munkaterület ablaka alatt az Output ablak foglal helyet. Ide érkeznek a rendszerüzenetek. Az ablakrendszer tetején található a menü és eszközsor.

A Visual Studio szövegszerkesztője intelligens szövegszerkesztő, amely a kiválasztott programozási nyelvben való kódolást segíti, többek között code snippetek felkínálásával. Automatikusan jelzi a kódolás közben elkövetett egyszerűbb hibákat is. Tesztelést támogató lehetőségeket biztosít, valamint a Debug szolgáltatással a hibák megtalálását segíti.

2. Az objektumorientáltság alapjai

Az objektumorientált paradigmával alapjaival a bevezető fejezetben már megismerkedett az olvasó. Itt az ideje, hogy megtanuljuk az alapelvek alkalmazását a C# programunkban is.

Képzeljük el, hogy azt a feladatot kaptuk, hogy készítsünk alkalmazást az önkormányzat számára, amely a bérbe adott lakások nyilvántartását végzi. Az alkalmazásnak tárolnia kell a lakások adatait és ki kell tudnia számítani az önkormányzat bérleti díjakból származó havi bevételét. A lakásokat jellemzi a helyrajzi száma, a területe, a komfortfokozata, a címe, havi bérleti díja, valamint a bérleti neve. A bérleti díj értéke minden lakásnál egyformán 800 Ft/m². Amelyik lakás komfortfokozata nem összkomfortos, a bérleti díjból 5% kedvezményt kap. Mindegyik lakás bérleti nevét, címét, bérleti díját ki kell tudni írni a képernyőre.

Belátható, hogy nem egyszerű feladattal van dolgunk. Az önkormányzatnak sok lakása van, és lakásonként több adatot kell tárolnunk. Hogyan kezelhetjük ezeket az adatokat? Érezhető, hogy a feladatot egyszerű változók bevezetésével nem tudjuk megoldani. Látható, hogy a nyilvántartandó adatok között szoros kapcsolat van, hiszen egy-egy konkrét lakásra vonatkoznak, tehát logikailag egy egységbe foglalhatók.

Gondolhatunk arra, hogy egy vektorba foglaljuk az egybe tartozó adatokat, de rögtön abba az akadályba ütközünk, hogy a tömb elemei csak azonos típusúak lehetnek, márpédig a lakás jellemzőit többféle adattípus írja le. Létrehozhatunk külön tömböt a helyrajzi számok, a lakásterületek, a komfortfokozat, a bérleti nevének stb. tárolására, de ezeknek a párhuzamos tömböknek a kezelése nagyon nehézkes lenne.

Jó lenne, ha a C# programozási nyelvben nemcsak integer, float stb. alaptípusokkal dolgozhatnánk, hanem tetszőleges típusokat, például lakástípust is használhatnánk. Tehát szükségünk van olyan **speciális változóra**, amely *lakás típusú*, és a lakásra vonatkozó összes adatot együtt tárolja, és még lehetőséget biztosít arra is, hogy a *tárolt adatokon műveleteket végezzünk*, esetünkben például a bérleti díjat ki tudjuk számolni. Ilyen speciális típus létrehozására van lehetőségünk, s ez a **speciális változó típus** lesz az **osztály**. A változó értéke pedig az adott **osztály példánya** lesz. Ezt a példányt **objektumnak** nevezzük.

Azzal, hogy létrehozunk egy osztályt, egy speciális típust hozunk létre, így az alaptípusokon kívül, mint az int, double stb., egy új típus is rendelkezésünkre áll. Annyiban többet nyújt az új speciális típus az alaptípusoknál, hogy nemcsak egy elemi értéket tudunk benne tárolni és kezelní, hanem egy olyan objektumot, amely magában foglalja az adott típus adattagjait és metódusait.

2.1. Elméleti háttér (Szendrői E.)

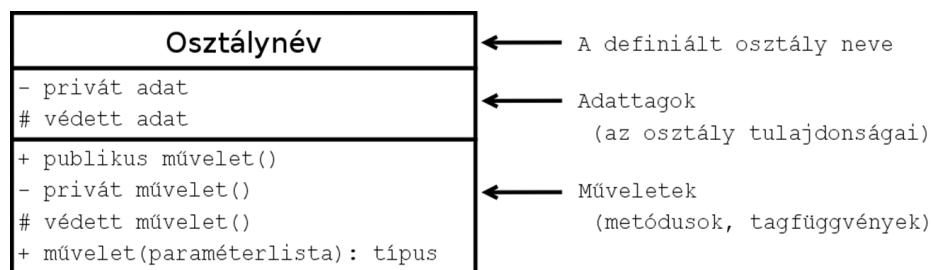
A közös tulajdonságokkal és viselkedéssel leírható **objektumok halmaza az osztály**. Az osztály egy olyan adatszerkezet, amely valamely entitásra vagy fogalomra jellemző logikailag összetartozó adatokat képes tárolni és azokon műveleteket tud végrehajtani. Az osztály adattagokat vagy más szóval mezőket és tagfüggvényeket (metódus) tartalmaz.

Adattagok: az osztályra vagy az osztály példányára jellemző adatokat tárolnak. Az osztály a valós világban létező objektumának tulajdonságait modellezi. Az adattagok aktuális értéke az objektum pillanatnyi állapotát tükrözi. A C# programozási nyelvben az adattagot típus és név megadásával definiáljuk.

A **tagfüggvények** vagy metódusok végrehajtják a bennük megadott utasításokat. Az objektumok viselkedését, műveleteit írják le.

Egy probléma megoldása során elemzéseket végzünk, megpróbáljuk megtalálni az összefüggéseket, s a probléma jobb megértéséhez és megoldásához modelleket alkotunk. A modellalkotás során tervezzük meg az alkalmazás struktúráját, az osztályokat és együttműködésüket. A jó modellekben bármilyen objektumorientált nyelvű alkalmazás készíthető. A modellek megfogalmazhatók szövegesen is, de sokkal könnyebb megérteni, ha grafikusan jelenítjük meg.

Az objektumorientált fejlesztés grafikus modellező eszköze az **UML szabványos modellező nyelv**. Többféle diagramtípust, modellt készíthetünk a segítségével, mint például osztály-, aktivitás-, szekvencia-, állapotdiagram stb. Programjaink tervezéséhez elsősorban az **osztálydiagramokra** lesz szükségünk, hiszen ezek az alkalmazás szerkezetét tükrözik. Az osztálydiagramon az osztályokat egy *három részre osztott téglalap* szimbolizálja, melyben az **osztály nevét**, az **adattagokat**, valamint a **metódusokat** tüntetjük fel. Az adattagok és metódusok hozzáférésmódját külön szimbólumokkal (-, +, #) jelöljük.

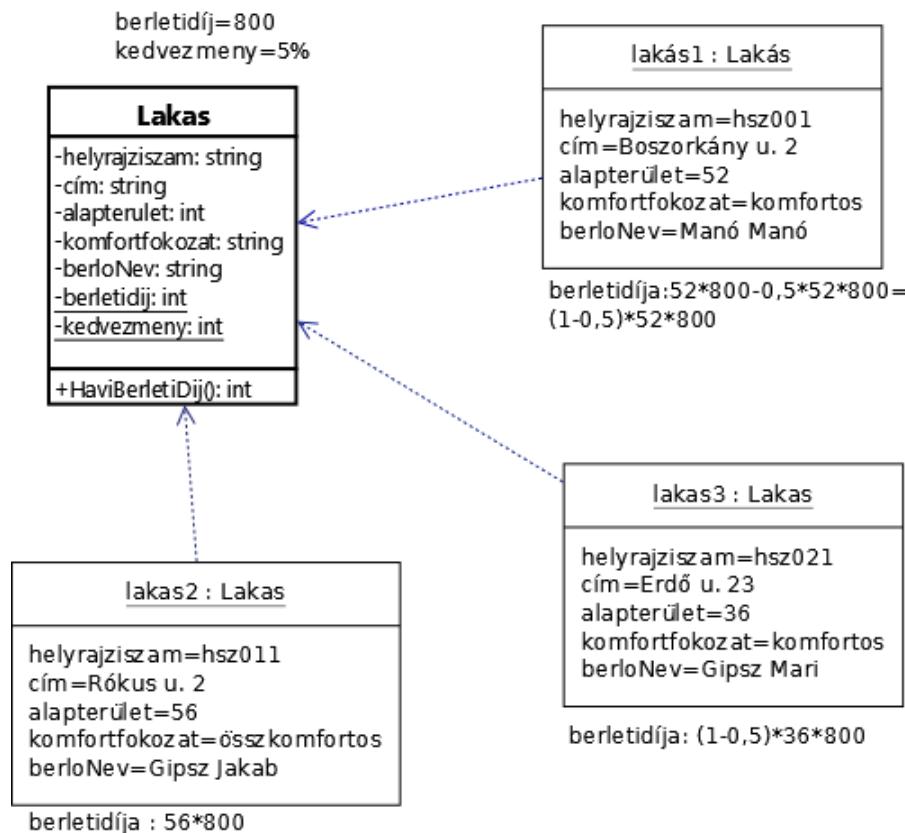


A 8. ábrán a fent vázolt feladat osztálydiagramja látható, kiegészítve az osztályból létrehozott objektumpéldányokkal is. Látható, hogy a feladatmegoldásunkban lesz egy **Lakas** osztály, és annak majd különböző példányait hozzuk létre, s végezzük el rajtuk a különböző műveleteket. Ezek a példányok a Lakas típusú (lakas1, lakas2, lakas3) objektumok.

Egy osztály adattagjait az **egységbézárás elve** alapján nem láthatjuk a külvilágóból, a külső osztályokból. Ezt jelzi a **private** hozzáférés-módosító, aminek az UML diagramon a – jel a jele. Az osztályokra jellemző az **információelrejtés** elve, ami azt jelenti, hogy az osztály adattagjaival és metódusaival egy a külvilágktól elzárt egységet képez, kívülről csak akkor férhetünk hozzá, ha ezt megengedjük. A HaviBerletiDij() metódus segítségével számoljuk ki a lakásonként fizetendő bérleti díjat az adattagok értékének ismeretében. A metódus a külső osztályok bármelyikéből hívható, **public** hozzáférés módosítóval rendelkezik, melynek UML jelölése a + jel.

Az ábrán van két adattag, amelyek neve alá van húzva. Ez a két adattag a bérleti díj és a kedvezmény százalékos mértéke. Mivel ezek több objektumpéldányra vonatkoznak, felesleges értéküket minden objektumnál tárolni és megadni, ezek az osztály minden

példányára egyformán érvényesek. Az ilyen adattagokat a **static** módosítóval jelezzük, és az UML ábrán aláhúzással jelöljük.



8. ábra A Lakas osztály diagramja

Osztály létrehozása C#-ban

Egy futó C# program egymással kommunikáló, együttműködő objektumok halmaza, amely objektumok különböző osztályok példányai.

A C# program osztályok sokasága. Az osztályokat általában külön fájlban hozzuk létre. A C# programban osztályt a **class** kulcsszó segítségével deklarálhatunk. Az **osztály referenciatípus**. Alapértelmezett láthatósági módosítója: **internal**. Az osztálynév minden nagybetűvel kezdődik.

Az osztály definiálásakor meg kell adnunk az osztály *nevét*, majd kapcsos zárójelek között az osztály törzsét, melyben deklarálni kell az *adattagokat*, *konstruktorkat* és *metódusokat*. Ennek formája a következő:

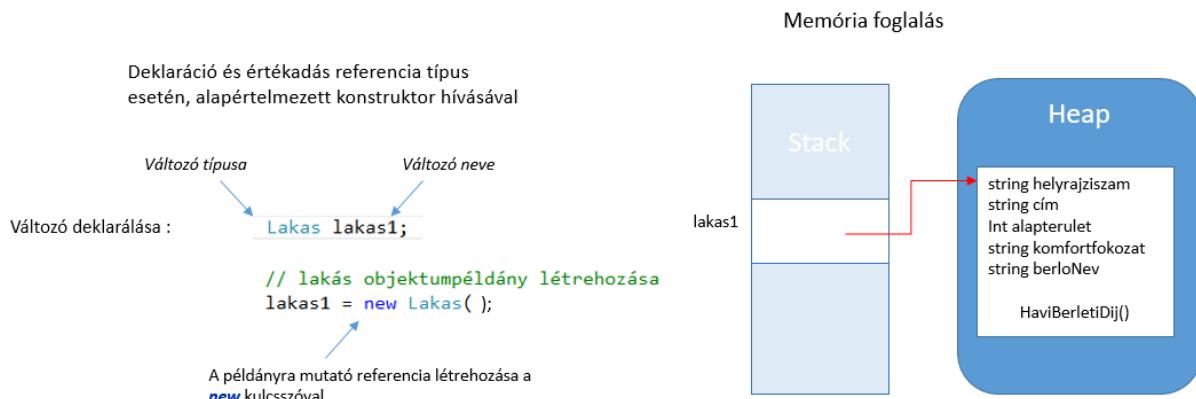
```

class Sajatosztaly
{
    adattagok (mezők);
    konstruktor(ok);
    metódus(ok);
}

```

Osztálytípusú változó és objektum létrehozása

Miután definiáltuk az osztályt, objektumpéldányokat hozhatunk létre. Az osztály, ahogy korábban már említettük, referenciátípus, ami azt jelenti, hogy minden adatokra való hivatkozás megadására, minden adatok számára memória helyet kell biztosítani. Az adatokra hivatkozást egy osztálytípusú változóban tároljuk. Az osztályból a **new** operátorral tudunk egy objektumpéldányt készíteni.



9. ábra Objektum létrehozása

A 9. ábrán láthatjuk, hogy létrejött a Stack memóriában egy *lakas1* nevű *Lakas* típusú változó, amely a *Lakas* osztály egy példányára mutat. A *lakas1* egy referencia, melyet fel tudunk használni a *Lakas* osztály egy objektumához (amely a heap memóriában van) tartozó adattagok, metódusok elérésére, a példány neve és az elérni kívánt adattag vagy metódus nevének felhasználásával. Például *lakas1.alapterulet* vagy *lakas1.HaviBerletiDij()*.

Minden esetben, amikor példányosítunk (a **new** hívásakor) egy speciális metódus, a **konstruktur** fut le. **A konstruktur nevénél meg kell egyeznie az osztály nevénél és nem lehet visszatérési értéke.**

Minden osztály rendelkezik egy alapértelmezett konstruktőrrel, még akkor is, ha mi egyetlenegy konstruktort sem definiáltunk az osztályban. Ezt a konstruktort default (azaz paraméter nélküli) konstruktornak hívjuk. A default konstruktur automatikusan jön létre és a jelen esetben a következő formájú:

```
// alapértelmezett konstruktur
public Lakas()
{ }
```

Látható, hogy ez egy metódus, *public* hozzáféréssel, nincs paramétere és visszatérési értéke, sőt a metódus törzse sem tartalmaz utasításokat. Az alapértelmezett (default) konstruktur legelőször meghívja a saját ősosztályára alapértelmezett konstruktőrét. Ez a *System.Object* konstruktora. (Korábban már említettük, hogy minden osztálynak az őse az *Object* osztály.) Az alapértelmezett konstruktur létrehoz egy objektumpéldányt, s helyet foglal

számára a memóriában, majd megtörténik az adattagok inicializálása, azaz kezdőértéket kapnak. Az adattagok automatikusan a nekik megfelelő nulla vagy *null* értékre inicializálódnak.

Egy osztály példányosításához legalább egy public hozzáférés-módosítóval ellátott konstruktora van szükség, egyébként nem fordul le a program.

Ha egy osztály tartalmaz paraméteres konstruktort, akkor az alapértelmezett (default) konstruktur nem jön létre automatikusan. Ha mégis szükségünk van rá, explicit módon definiálnunk kell, a fent látható módon.

Egy osztálynak bármennyi paraméteres konstruktora lehet, azonban paraméterlistájuk nem lehet azonos. Ha több konstruktora van egy osztálynak, akkor a megadott paraméterek számától függ, hogy melyik kerül meghívásra (szignatúra).

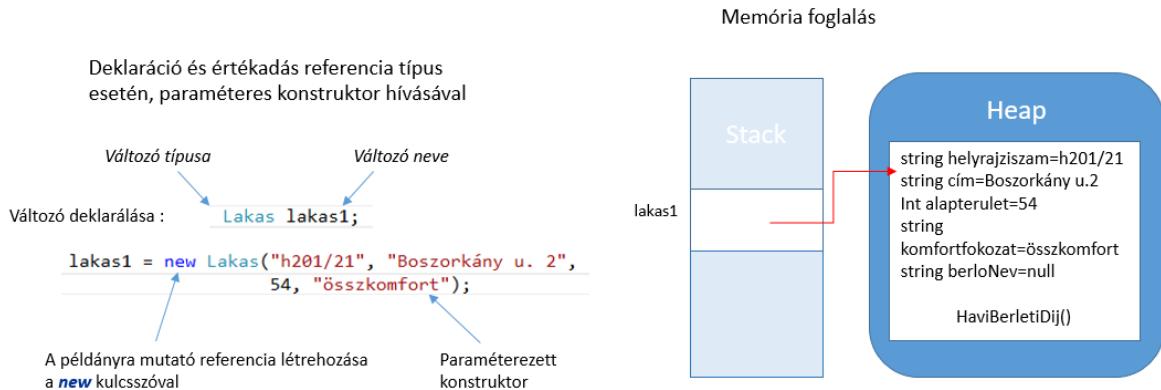
Mikor használunk paraméteres konstruktort? Ha azt szeretnénk, hogy az objektumpéldány létrejöttekor egyes adattagoknak legyen általunk beállított kezdőértéke. Leggyakoribb eset az, amikor már az objektum létrejöttekor szükségünk van arra, hogy bizonyos adattagok konkrét értékkel rendelkezzenek, sőt a továbbiakban ezeket az értékeket nem is szabad megváltoztatni. Ilyenkor a legjobb, ha paraméteres konstruktort írunk és a paraméterek segítségével adjuk meg az adattagok értékét. A Lakas osztályunk esetében ilyen adattag a *helyrajziszam*, az *alapterulet*, a *cím* és a *komfortfokozat*. Ezek már ismertek az objektumpéldány létrehozatalakor és nem változnak. (Persze, fel lehet újítani egy lakást és így megváltozhat a komfortfokozata, átnevezheti az önkormányzat az utcát, de most ettől eltekintünk. Egyébként látni fogjuk, hogy ha megengedjük, az adattagok értéke később is módosítható.) Lássunk egy példát a paraméteres konstruktorra!

```
// paraméteres konstruktur
2 references
public Lakas(string helyrajziSzam, string cim, int alapterulet,
    string komfortfokozat) {

    this.helyrajziSzam = helyrajziSzam;
    this.cim = cim;
    this.alapterulet = alapterulet;
    this.komfortfokozat = komfortfokozat;
}
```

A konstruktur paramétereivel rendelkezik, és a hívásakor megadott értékekkel tudjuk majd beállítani az adattagok értékét. A konstruktur törzsében látható értékkedő utasításokban az értékkedés jel bal oldalán a **this** kulcsszóval hivatkozunk az aktuális objektumpéldányra, majd a pont (.) operátort követően adjuk meg az adattag nevét. Az értékkedés jobb oldalán, a paraméterlistán szereplő paraméterek láthatók.

A következő ábrán (10. ábra) az objektum létrehozása és helyfoglalása látható paraméteres konstruktur használatával.



10. ábra Objektum létrehozása paraméteres konstruktőrrel

Látható, hogy az érték szerinti paraméterátadás által történik meg az adattagok értékének inicializálása. A konstruktur hívásakor megadott, aktuális paraméterlistán lévő értékeket átadjuk a formális paraméterekeknek, majd ezekkel a kapott értékekkel tudunk értéket adni a konstruktur törzsében felsorolt adattagoknak, jelen esetben a *helyrajziSzam*, *cím*, *alapterulet* és *komfortfokozat* változóknak.

```
// lakás objektumpéldány létrehozása
lakas1 = new Lakas("h201/21", "Boszorkány u. 2", 54, "összkomfort");

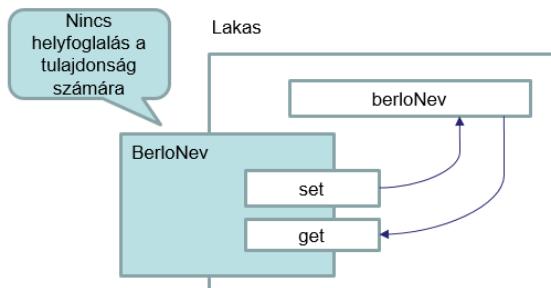
public Lakas(string helyrajziSzam, string cim, int alapterulet,
    string komfortfokozat) {
    this.helyrajziSzam = helyrajziSzam; // "h201/21"
    this.cim = cim; // "Boszorkány u. 2"
    this.alapterulet = alapterulet; // 54
    this.komfortfokozat = komfortfokozat; // "összkomfort"
}
```

Tulajdonság

Korábban említettük, hogy az egységbezárás elve alapján az adattagoknak *private* módosítójuk van, tehát külső osztályokból nem láthatóak, így nem is tudnak értéket kapni, ezáltal nem módosíthatók, de még az értéküket sem tudjuk kiolvasni. Jelen feladatunkban, például, ha megváltozik a lakás bérlöje, nem tudjuk a bérlö következőképpen adhatunk meg:

```
private string berloNev;

// tulajdonság (Property)
2 references
public string BerloNev {
    get { return berloNev; }
    set { berloNev = value; }
}
```



A tulajdonságok speciális metódusok. Public módosítóval rendelkeznek, amely azt jelenti, hogy külső osztályokból elérhetők, ellentétben a *private* adattagokkal. A módosítót követi a tulajdonság típusa, ami azonos az elérni kívánt adattag típusával. Ezután következik a tulajdonság neve, ami azonos az adattag nevével, azzal a különbséggel, hogy konvenció szerint **nagy kezdőbetűvel** írjuk. A tulajdonság definiálása után kapcsos zárójelek között szerepel a **get** és **set** blokk, melyek biztosítják az adattag lekérdezési és módosítási lehetőségét. Az előbbi a tulajdonság mögött lévő mező értékét adja vissza, az utóbbi pedig értéket ad neki. minden esetben, amikor az adott tulajdonságnak értéket adunk, egy „láthatatlan” paraméter jön létre, aminek a neve **value**, ez fogja tartalmazni a tulajdonság új értékét. Fontos tudni, hogy a tulajdonság deklarálásakor nincs memóriahely-foglalás, hiszen a tulajdonság nem változó, hanem ahogy már említettük, egy speciális metódus.

A get és set blokk elérhetősége nem kell, hogy azonos legyen, de a get blokknak mindenképpen publikusnak kell lennie.

```
public string BerloNeve {
    get { return berloNeve; }
    private set { berloNeve = value; }
}
```

Lehetőség van arra is, hogy csak a get vagy a set blokkot adjuk meg, ekkor **csak írható** vagy **csak olvasható** tulajdonságról beszélünk.

```
public int Alapterulet {
    get { return alapterulet; }
}
```

A C# 3.0 egyik újítása az ún. **automatikus tulajdonság** bevezetése volt. Az automatikus tulajdonság lényege, hogy nem kell létrehoznunk sem az adattagot, sem a teljes tulajdonságot, a fordító mindenkorrel legenerálja helyettünk:

```
public string BerloNeve
{
    get ;set ;
}
```

A fordító foglal memóriahelyet

A fordító automatikusan létrehoz egy private elérésű, string típusú „berloNeve” nevű adattagot és elkészíti hozzá a get /set blokkot is. Azonban a fordítás pillanatában ez a változó még nem létezik, vagyis közvetlenül nem hivatkozhatunk rá például a konstruktorban.

A tulajdonságok nemcsak arra alkalmasak, hogy kiolvassák vagy módosítsák a *private* adattagok értékét. Lehetőséget biztosítanak arra, hogy műveleteket is végrehajtsunk a get és set blokkokban. Például ellenőrzött módon adhatunk értéket a tulajdonságon keresztül egy adattagnak. A következő példa ezt mutatja. Ha véletlenül a kedvezménynek 100-nál nagyobb értéket adnánk, azt automatikusan 100-ra módosítja. Egyébként pedig a *value*-ban található értéket kapja a tulajdonság, s a mögötte lévő private adattag.

```
public static int Kedvezmeny {
    get { return kedvezmeny; }
    set { kedvezmeny = value > 100 ? 100:value; }
}
```

Statikus (objektumfüggetlen) mezők, statikus tulajdonságok, statikus metódusok

Az osztály adattagjainak egy része az objektumpéldányokhoz, még más mezők az egész osztályhoz és nem az egyes példányokhoz köthetők. Az osztály szintű mezők olyan jellegű adatokat hordoznak, amelyet elég egyetlen példányban tárolni, mivel ezen adat minden objektumpéldány esetén amúgy is ugyanolyan értéket hordozna. Ez memóriatakarékos megoldás. Az ilyen jellegű mezőket a **static** kulcsszóval kell megjelölni.

A megértés megkönnyítésére gondoljunk arra például, hogy egy konkrét autótípus modellezése esetén az adott típus minden egyes példányához (konkrét autók) ugyanakkora benzintankméret tartozik, ezért ezen érték közös a példányok között. Ugyanakkor az, hogy mi a rendszám, és mennyi benzin van éppen e pillanatban a tankban, az már példányonként eltér. Tehát a tankmérőt az adott autótípus-osztályra jellemző, ennek megfelelően statikus mező lesz, míg az üzemanyag mennyisége, valamint a rendszám az adott konkrét autóobjektumot jellemzi, tehát példánymező lesz.

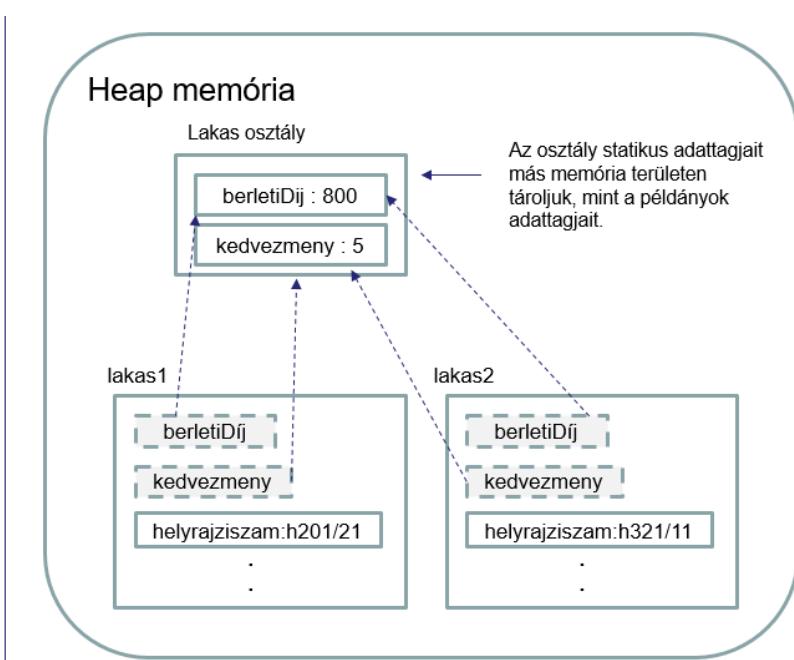
Ha a lakásbérleti példánkat tekintjük, ott is szükségünk van statikus mezőkre, ahogy korábban már szó volt róla. A négyzetméterenkénti bérleti díj minden lakásra egyformán 800 Ft. Hasonlóan, a kedvezmény mértéke is azonos minden lakás esetében, bár az összkomfortos lakásokra nem jár kedvezmény.

A **static** módosítóval deklarált mezők a memoriában példányuktól függetlenül, egyszer szerepelnek. Már akkor is benne van a bennük tárolt érték a memoriában, amikor még az osztályból egyetlen példányt sem készítettünk el.

A hagyományos (példány-) adattagok és metódusok objektumszinten léteznek, azaz minden objektumpéldány saját tagpéldánnyal rendelkezik.

```
//statikus mezők
private static int kedvezmeny;
private static int berletiDij;

// példány tagok
private string helyrajziszam;
private string cim;
private int alapterulet;
private string komfortfokozat;
private string berloNev;
```



11. ábra Statikus és példányadattagok

Statikus tag neve konvenció szerint nagybetűvel kezdődik. A statikus tagokhoz az osztály nevén (és nem egy példányán) keresztül, férünk hozzá.

A **tulajdonságokat** is definiálhatjuk **statikusnak**. Hasonlóan a statikus tagokhoz, objektumpéldány létrehozása nélkül dolgozhatunk velük. Statikus tulajdonság nem fér hozzá példánytagokhoz. Külső osztályokból az osztály nevén és a pont (.) operátoron keresztül hivatkozhatunk rájuk. Statikus *private* adattagok eléréséhez ugyanolyan típusú statikus tulajdonságokat kell definiálni. Példa a lakás osztályból:

```
//statikus mezők
private static int kedvezmeny;
private static int berletiDij;

// statikus tulajdonságok
1 reference
public static int BerletiDij {
    get { return berletiDij; }
    set { berletiDij = value; }
}
1 reference
public static int Kedvezmeny {
    get { return kedvezmeny; }
    set { kedvezmeny = value; }
}
```

Statikus és példánytagok, tulajdonságok elérése külső osztályokból

Korábban már szó volt róla, hogy egy osztály *példányadattagaihoz* az objektumpéldány létrehozása után férhetünk hozzá. Természetesen csak publikus adattagokat érhetünk el külső osztályból. Az egységbézárás elvének betartása révén publikus adattagokat nem definiálunk, helyette tulajdonságokat hozunk létre az adattagok elérésére. A hivatkozás az objektum nevének megadásával, majd a pont (.) operátor, s ezt követően az adattag vagy tulajdonság nevének a megadásával lehetséges.

Statikus tagokra (mezőkre) vagy tulajdonságokra az osztálynév, majd a pont (.) operátor és a mező vagy tulajdonság név megadásával hivatkozhatunk.

Az alábbi példa a lakásnyilvántartó alkalmazásunk vezérlő osztályából való hivatkozásokat mutat a Lakas osztály példány és statikus tulajdonságaira.

```
// statikus tulajdonságok megadása
Lakas.BerletiDij = 800;
Lakas.Kedvezmeny = 5;

// lakás objektumpéldány létrehozása
// konstruktor hívása
Lakas lakas1 = new Lakas(hrsz, cim, terulet, komfortfok);

// értékadás példány tulajdonságának
lakas1.BerloNev = "Gipsz Jakab";
```

Statikus (objektumfüggetlen) metódusok

A statikus tagokhoz és tulajdonságokhoz hasonlóan, objektumpéldány létrehozásától függetlenek, tehát objektum létrehozása nélkül is meghívhatók. Statikus függvények nem férnek hozzá közvetlenül a példánytagokhoz.

A statikus metódusoknak, hasonlóan a példánymetódusokhoz, lehet paraméterlistájuk és visszatérési értékük is. **Statikus metódusból csak statikus metódus hívható**. Akkor használjuk, ha nem egy objektumpéldány állapotának megváltoztatása a cél, hanem az osztályhoz köthető művelet elvégzése a feladat. A „leghíresebb” statikus metódus a Main(). Külső osztályból statikus metódus hívása az osztály neve, majd a pont operátor és a metódus nevének és paramétereinek megadásával történik.

Statikus konstruktur

Egy konstruktort deklarálhatunk statikusnak. Míg a példánykonstruktur létrehozza és inicializálja az osztály új objektumpéldányait, addig a statikus konstruktur az osztály statikus tagjainak beállításáért felel. Általánosságban azt mondhatjuk, hogy a statikus konstruktur az osztály statikus mezőit inicializálja. Az osztály mezőket a rájuk való hivatkozás és bármilyen objektumpéldány létrehozása előtt kell inicializálni. A statikus konstruktorkor annyiban hasonlítanak a példánykonstruktorkhoz, hogy az ő nevük is az osztály nevével azonos, és nem lehet visszatérési értékük. A statikus konstruktur abban különbözik a példánykonstruktortól, hogy nem lehet paraméterlistája, valamint csak egy statikus konstruktora lehet egy osztálynak. A statikus konstruktort a static kulcsszóval jelezzük. A statikus konstruktorthoz nem lehet módosítót (pl. private, public stb.) hozzárendelni.

Fontos tudni, hogy egy osztályban definiálhatunk statikus és példánykonstruktort is. A statikus konstruktur nem fér hozzá az osztály példánytagjaihoz és nem használhatjuk a törzsében a **this** hivatkozást. (Ez logikus, hiszen a **this** az aktuális objektumra mutat.) A statikus konstruktur explicit módon nem hívható, automatikusan történik a hívása, mielőtt egy statikus mezőre szeretnénk hivatkozni, vagy mielőtt egy objektumpéldányt létrehoznánk. Alábbi példa statikus konstruktort használ egy privát statikus mező értékének inicializálására.

```
class VeletlenSzamOszt
{
    private static Random RandErtek;    //privát statikus mező
    static VeletlenSzamOszt()        //statikus konstruktur
    {
        RandErtek = new Random();    //Randertek inicializálása
    }
    public int VeletlenSzam()
    {
        return RandErtek.Next();
    }
}
```

Az osztály felhasználása egy egyszerű programban, véletlen szám előállítására:

```
  REFERENCES
class Program
{
    REFERENCES
    static void Main(string[] args)
    {
        VeletlenSzamOszt a = new VeletlenSzamOszt();
        VeletlenSzamOszt b = new VeletlenSzamOszt();

        Console.WriteLine("következő véletlen szám: {0}", a.VeletlenSzam());
        Console.WriteLine("következő véletlen szám: {0}", b.VeletlenSzam());
    }
}
```

Statikus osztály

A statikus osztály minden tagja statikus. Statikus osztályokat az azzal a céllal hozunk létre, hogy egységbe foglaljuk adatok és függvények olyan csoportját, amelyekre nincsenek hatással objektumpéldányok. Különböző műszaki számításokhoz szükségünk van matematikai függvények, állandók, egyéb számítások végrehajtására. Ilyenkor kézenfekvő, hogy létrehozunk a matematikai összefüggések számára egy könyvtárat (osztályt), amely matematikai függvényeket és állandókat tartalmaz.

Néhány fontos tudnivaló a statikus osztályokról:

- A **static** kulcsszóval jelezzük az osztálydefiníció során, hogy az osztály statikus.
- Statikus osztály minden tagja csak statikus lehet.
- Az osztálynak lehet egy statikus konstruktora, de érthető módon, példánykonstruktora nem lehet.
- Statikus osztály implicit módon lezárt, ami azt jelenti, hogy nem hozhatunk létre belőle leszármazott osztályt (erről a 4. fejezetben lesz szó).
- Statikus osztályból objektumpéldány nem hozható létre.
- Statikus osztály tagjait az osztály nevével és a tag nevével érhetjük el.

A .NET Framework alap osztálykönyvtára (Base Class Library) több statikus osztályt tartalmaz, amelyek metódusaira, függvényeire a programjainkból az osztály nevével és a tag vagy függvény nevével hivatkozhatunk. Ilyen statikus osztály például a **Console** osztály. Ahhoz, hogy a metódusait meghívassuk, nem kell objektumpéldányt létrehozni, hanem közvetlenül az osztály nevét és a metódus nevét kell megadnunk, például `Console.WriteLine()`.

Hasonlóan statikus osztály a **Math** matematika osztály. A PI értékére, mint statikus konstansra, az alábbi módon hivatkozhatunk:

`Math.PI`, ami double típusú konstans.

Az alábbi táblázat a Math osztály néhány statikus metódusát tartalmazza:

Math osztály statikus függvényei	Matematikai függvény
Math.Sin(x)	$\sin(x)$
Math.Cos(x)	$\cos(x)$
Math.Tan(x)	$\operatorname{tg}(x)$
Math.Exp(x)	e^x
Math.Log(x)	$\ln(x)$
Math.Sqrt(x)	\sqrt{x}
Math.Abs(x)	$ x $
Math.Pow(x,y)	x^y
Math.Round(x)	Kerekítés

A Console és a matematika osztály statikus metódusain túl, más statikus metódusokat is gyakran kell használnunk az alkalmazásainkban. Ilyen a **Parse()** és a **TryParse()** statikus metódus is. Ezekre a metódusokra szinte minden beolvasást követően szükségünk van. A billentyűzetről történő beolvasás ugyanis karaktersorozatot eredményez, s ha valójában numerikus adatot akarunk beolvasni, azt át kell alakítani a megadott numerikus típusra. Mindegyik numerikus típuson meghívhatjuk a Parse() vagy a TryParse() metódusokat, a numerikus típus nevének megadását követően. Például:

```
float sugar, magassag, terfogat;
Console.WriteLine("Kérem a henger sugarát:");
sugar=float.Parse(Console.ReadLine());
Console.WriteLine("Kérem a henger magasságát:");
magassag = float.Parse(Console.ReadLine());
```

A TryParse() statikus metódus logikai értékkel tér vissza, ha sikerül a beolvasott karaktersorozatot az adott típusra konvertálni, s az **out** paraméterben adja vissza a beolvasott értéket. Ellenőrzött beolvasásra nagyon jól alkalmazható. Az alábbi példán egy ilyen beolvasás látható.

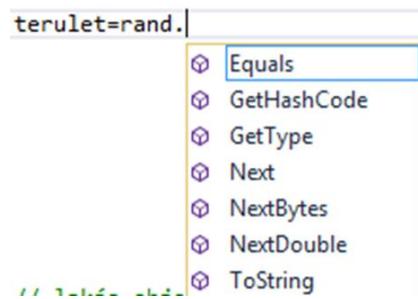
```
int szam, oszto;
Console.WriteLine("Kérek egy pozitív számot: ");
while (!int.TryParse(Console.ReadLine(), out szam) || szam <= 0)
{
    Console.WriteLine("Hibás adat, kérem újra! ");
}
```

A Random osztály

Bár nem statikus a Random osztály, de mindenéppen érdemes külön megemlíteni. A **Random** osztály metódusai nem statikus metódusok, ezért elérésükhez objektumpéldányt kell létrehozni. Formája:

```
Random rand = new Random();
```

A Random osztály metódusai a következő ábrán láthatók:



Az osztály Next() metódusának hívásával minden újabb véletlen egész számot kapunk. A Next() metódus többféleképpen paraméterezhető, ezt túlterhelt (overload) metódusnak nevezzük. A szignatúra alapján tudja a fordító, hogy melyik változatot kell meghívni. A különbség az egyes hívási formák között, hogy milyen intervallumban helyezkedik el az eredmény:

```
int randSzam;
randSzam= rand.Next();
// intervallum: [0, 2147483647]
```

Ha a Next() metódusnak egy nem negatív egészértéket adunk meg bemenő paraméterként, akkor a 0 és a megadott érték között generál egy véletlen számot. Az intervallum alulról zárt, felülről nyitott, így a megadott érték-1 lesz a legnagyobb érték, ami generálódhat.

```
randSzam = rand.Next(128); // intervallum [0, 128)
```

Két egész típusú bemenő paraméterrel is meghívható a Next() metódus. Ekkor a két érték közötti véletlen értékkel tér vissza.

```
randSzam = rand.Next(20,101); // intervallum [20, 101)
```

Ha a NextDouble() metódust használjuk, akkor a [0.0, 1.0) félgyűjtő nyílt intervallumban kapunk double típusú véletlen számot.

```
Random random=new Random();
// double típusú véletlen szám
double vszam;
vszam=random.NextDouble();
```

Konstruktörinicializáló

Korábban már láttuk, hogy egy osztálynak több paraméteres konstruktora lehet, ha a paraméterek száma eltérő. (Overloading)

Nézzünk egy példát: Ingatlanokat akarunk nyilvántartani. Jellemzi az ingatlanokat a típusuk (ház, lakás, nyaraló), a szobák száma, valamint, hogy van-e garázs és kert. Létrehozunk egy **Ingatlan** osztályt és több konstruktort írunk, hogy könnyen létre tudjuk hozni a különböző objektumokat, hiszen lehet kert nélküli, garázs nélküli az ingatlan, vagy pont valamelyikkel vagy akár mindegyikkel rendelkezik. Az Ingatlan osztály kódja a konstruktorokkal:

```
class Ingatlan
{
    private string tipus; // lehet ház, lakás, nyaraló
    private int szobakSzama;
    private bool garazsVan;
    private bool kertVan;

    public Ingatlan(string tipus, int szobakSzama) {
        this.tipus = tipus;
        this.szobakSzama = szobakSzama;
    }
    public Ingatlan(string tipus, int szobakSzama, bool garazsVan) {
        this.tipus = tipus;
        this.szobakSzama = szobakSzama;
        this.garazsVan = garazsVan;
    }
    public Ingatlan(string tipus, int szobakSzama, bool garazsVan, bool kertVan) {
        this.tipus = tipus;
        this.szobakSzama = szobakSzama;
        this.garazsVan = garazsVan;
        this.kertVan = kertVan;
    }
}
```

The code shows three constructor definitions for the **Ingatlan** class. Three arrows point from the start of each constructor definition to a single label "Konstruktörök" (Constructors) located to the right of the class definition.

Lehetőség van arra, hogy egy konstruktur meghívja a saját osztályának egy másik konstruktörét a **this** kulcsszó alkalmazásával a konstruktur definíciójában. Azt a konstruktort hívja meg, amelyiknek a szignatúrája illeszkedik a definícióban megadott szignatúrára.

Például létrehozhatunk egy default konstruktort, amely egy paraméteres konstruktort hív meg, a megadott alapértékekkel.

```
class Ingatlan
{
    private string tipus; //lehet ház, lakás, nyaraló
    ...

    public Ingatlan(string tipus, int szobakSzama, bool garazsVan, bool kertVan) {
        this.tipus = tipus;
        this.szobakSzama = szobakSzama;
        this.garazsVan = garazsVan;
        this.kertVan = kertVan;
    }

    // Default konstruktur, amely egy 3 szobás garázsos, kertes házat hoz létre
    public Ingatlan() : this("ház", 3, true, true) {
    }
}
```

The code shows a default constructor definition that initializes the object using the constructor defined above it. An arrow points from the start of this constructor definition to a label "Konstruktur inicializáló" (Constructor initializer).

Objektuminicializáló

Egy objektum létrehozása a konstruktur hívásával történik. Láttuk, a konstruktornak lehetnek paraméterei, amelyekkel az objektum adattagjainak kezdőértéket adunk az objektum létrehozásakor.

Az **Ingatlan** osztálynak több paraméteres konstruktora van, így több lehetséges ingatlanvariáció esetén könnyen létre tudjuk hozni a megfelelő ingatlanobjektumot. Nem biztos azonban, hogy minden lehetséges változatra készíthetünk konstruktort. Az Ingatlan osztályban arra az esetre nem készítettünk konstruktort, amikor a típus, szobaszám és a kert van, értékét akarjuk megadni az objektum létrehozásakor.

A probléma megoldására létrehozhatunk egy negyedik konstruktort, csakhogy a fordító hibát jelez. Miért? Azért, mert, ahogy az ábra is mutatja, a negyedik konstruktur szignatúrája teljesen megegyezik a harmadik konstruktoreval, ezt pedig a fordító nem engedi meg.

```
public Ingatlan(string tipus, int szobakSzama, bool garazsVan) {
    this.tipus = tipus;
    this.szobakSzama = szobakSzama;
    this.garazsVan = garazsVan;
}
public Ingatlan(string tipus, int szobakSzama, bool kertVan) {
    this.tipus = tipus;
    this.szobakSzama = szobakSzama;
    this.kertVan = kertVan;
}
```

Mi a megoldás?

Ilyen esetben az objektuminicializálót hívhatjuk segítségül, ami a C# 3.0 újítása. Az objektuminicializáló létrehoz egy objektumot a megadott konstruktur segítségével és ugyanabban az utasításban a megadott tulajdonságokat is inicializálja.

Formája: Az inicializálni kívánt tulajdonságokat és értéküket a konstruktur hívása után kapcsos zárójelek között kell megadni. Nézzük a megoldást:

Definiálunk egy tulajdonságot az osztályban.

```
public bool KertVan
{
    get { return kertVan; }
    set { kertVan = value; }
}
```

Ezután a kerttel rendelkező objektumpéldányt az alábbi utasítással objektuminicializálóval hozzuk létre.

```
// Objektum létrehozása paraméteres konstruktőrrel
// és objektum inicializálóval
Ingatlan azEnHazam = new Ingatlan("ház", 2) { KertVan = true };
```

Amikor objektuminicializálóval hozunk létre egy objektumot, először a megadott konstruktur fut le, majd az inicializáló hozzárendeli a megadott értékeket az adattagokhoz, tulajdonságokhoz.

Az objektuminicializáló felülírja a konstruktur által a *default* értékekkel inicializált adattagokat. Esetünkben a konstruktur *false* értéket rendel a *kertVan* és *garazsVan* adattagokhoz, majd az objektuminicializáló a *KertVan* tulajdonság értékének megadásával *true* értékre módosítja azt.

Osztályok együttműködése

Egy objektumorientált alkalmazás osztályok halmazából áll, amelyeket külön fájlokban hozunk létre, így másik alkalmazás készítéséhez is felhasználható. Lehetőség van egy fájlban több osztályt is definiálni, de akkor elveszíthetjük az újrahasznosítás lehetőségét, ami az objektumorientáltság egy jellemzője. Az objektumorientált alkalmazás az osztályok együttműködése révén valósítja meg feladatát.

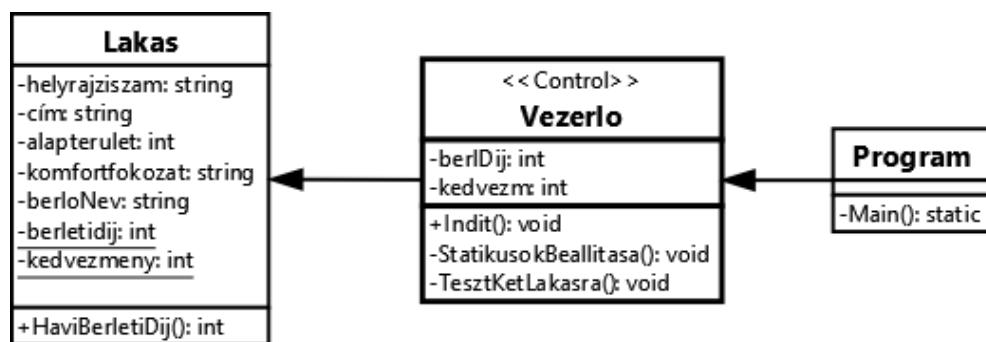
Korábban láttuk, hogy amikor egy új alkalmazást hozunk létre a Visual Studióban, automatikusan létrejön a Program.cs fájl és a Program osztály, benne a Main() metódussal, ami a programunk belépési pontja. A Main() metódus statikus, ami a korábban megtanultak alapján azt jelenti, hogy csak statikus metódust hívhatunk belőle. A Program osztálynak az a feladata, hogy elindítsa az alkalmazást. Ennél többet nem is bízunk rá. Ugyanakkor szükségünk van egy újabb osztályra, amely biztosítja számunkra az osztályok közötti kommunikációt, együttműködést, vezérli az alkalmazás működését.

Amikor azt szeretnénk, hogy egy osztály kapcsolatba kerüljön egy másik osztálytal, azt úgy valósítjuk meg, hogy létrehozunk egy objektumpéldányt a szükséges osztályból abban az osztályban, ahol használni szeretnénk az adott osztály szolgáltatásait. A létrejött objektumpéldányon keresztül tudjuk majd meghívni az osztályban definiált metódusokat, amelyek által az osztály végrehajtja a saját feladatát.

Az elmondottak alapján létre kell hoznunk egy vezérlő osztályt, melynek az lesz a feladata, hogy tartsa a kapcsolatot a felhasználóval, vagyis beolvasson adatokat, ha szükséges, és létrehozza a szükséges objektumpéldányokat, meghívja azok metódusát, hogy elvégezzék feladatukat. Tehát a mintafeladatunk **Lakas** osztályát a **Vezerlo** osztályban szeretnénk majd kezelni. A vezérlő osztály fogja kiírni a képernyőre a számítások eredményét. Hol fogjuk létrehozni a Vezerlo osztály példányát? A program belépési pontján, azaz a Program osztály Main() metódusában.

A **Vezerlo** osztályban létrehoztunk egy *Indit()* metódust, ennek a metódusnak a feladata, hogy minden további metódust meghívjon, és ezzel vezérelje a végrehajtás folyamatát.

Az alkalmazás véleges osztályszerkezete:



Összefoglalva: Az alkalmazás végrehajtása az indító osztályból (Program osztály belépési pontja (Main metódus)) indul. Szükségünk van egy újabb osztályra, amely vezéri az alkalmazás működését, az osztályok együttműködését. Ezt valósítja meg a *Vezerlo* osztály, amiből létrehozunk egy példányt az indító osztály Main() metódusában. Ettől kezdve a folyamatokat a vezérlő osztály példánya irányítja.

A vezérlő osztály kódja: (csak a teszt kedvéért van beolvasás helyett értékadás a kódban)

```

class Vezerlo
{
    int berlDij = 800, kedvezm = 5;
    1 reference
    public void Indit() {
        StatikusokBeallitasa();
        TesztKetLakasra();

        Console.ReadKey();
    }
    1 reference
    private void StatikusokBeallitasa() {
        // statikus tulajdonságok megadása
        Lakas.BerletiDij = berlDij;
        Lakas.Kedvezmeny = kedvezm;
    }
    1 reference
    private void TesztKetLakasra()
    {
        // lakas1 és lakas2 nevu változók deklarálása
        Lakas lakas1;
        Lakas lakas2;
        string hrsz, cim, nev, komfortfok;
        int terulet, bevetel = 0;
        // első lakás
        hrsz = "h201/21";
        cim = "Boszorkány u. 2";
        nev = "Gipsz Jakab";
        terulet = 54;
        komfortfok = "összkomfort";
        // lakás objektumpéldány létrehozása
        lakas1 = new Lakas(hrsz, cim, terulet, komfortfok);
        lakas1.BerloNev = nev;
        // második lakás
        hrsz = "h321/11";
        cim = "Rókus u. 2";
        nev = "Gipsz Vilma";
        terulet = 48;
        komfortfok = "komfort";
        // második lakásobjektum létrehozása
        lakas2 = new Lakas(hrsz, cim, terulet, komfortfok);
        lakas2.BerloNev = nev;
        // a lakások jellemzőinek kiíratása
        Console.WriteLine(lakas1);
        Console.WriteLine(lakas2);
        // önkormányzat bevételle a lakások bérbeadásából
        bevetel = lakas1.HaviBerletiDij() + lakas2.HaviBerletiDij();
        Console.WriteLine("\nAz önkormányzat havi bevételle: " + bevetel + " Ft.");
    }
}

```

A **Lakas** osztály teljes kódja:

```
class Lakas
{
    private string helyrajziSzam;
    private string cim;
    private int alapterulet; // felte tesszük, hogy egész érték
    private string komfortfokozat;
    private string berloNev;
    // statikus változók minden objektumpéldányra egyformán vonatkoznak,
    // az osztályra jellemzők
    private static int kedvezmeny;
    private static int berletiDij;
    // paraméteres konstruktur
    2 references
    public Lakas(string helyrajziSzam, string cim, int alapterulet,
        string komfortfokozat) {
        this.helyrajziSzam = helyrajziSzam;
        this.cim = cim;
        this.alapterulet = alapterulet;
        this.komfortfokozat = komfortfokozat;
    }
    // havi bérletidíj kiszámítása, figyelembe véve az esetleges kedvezményt
    3 references
    public int HaviBerletiDij() {
        if (komfortfokozat=="összkomfortos"){
            return alapterulet * berletiDij;
        }
        else{
            return alapterulet * berletiDij * (1 - kedvezmeny / 100);
        }
    }
    0 references
    public override string ToString() {
        return "A "+cim+" -ben lévő lakás bérlete: "+berloNev+"\n      a lakásért havonta "+this.HaviBerletiDij()+" Ft bérleti díjat fizet. \n" ;
    }
}
```

A fenti kódban szereplő `ToString()` metódusról kell még néhány szót ejteni. A `ToString()` metódus az **object** osztály metódusa, s mivel minden osztály az `object`ból származik, így megörökli a `ToString()` metódust is. Ebben a kiíratásban az `object`ben definiált `ToString()` metódusnak egy módosított változatát adtuk meg, ezt jeleztük az `override` kulcsszóval. Erről a témáról majd a 4. fejezetben lesz bővebben szó.

Ami még látható a `ToString()` törzsében, hogy egyetlen sztringbe fűztük össze a kiírandó információkat. A karaktersorozat összefűzését a `+` jellel jelezzük. A sztring kifejezésben hívtuk meg az aktuális objektum `HaviBerletiDij()` metódusát az érték kiszámítására a `this.HaviBerletiDij()` hivatkozással.

A **Lakas** osztály kódja a következő kód részletben folytatódik, amelyben az osztály tulajdonságai láthatók.

```

// tulajdonságok
0 references
public string HelyrajziSzam {
    get { return helyrajziSzam; }
}
0 references
public string Cim {
    get { return cim; }
}
0 references
public int Alapterulet {
    get { return alapterulet; }
}
0 references
public string Komfortfokozat {
    get { return komfortfokozat; }
}
2 references
public string BerloNev {
    get { return berloNev; }
    set { berloNev = value; }
}
// statikus tulajdonságok
1 reference
public static int BerletiDij {
    get { return berletiDij; }
    set { berletiDij = value; }
}
1 reference
public static int Kedvezmeny {
    get { return kedvezmeny; }
    set { kedvezmeny = value; }
}
}

```

A Program osztály kódja a következő:

```

class Program
{
    0 references
    static void Main(string[] args) {
// létrehozzuk a Vezerlo osztály egy példányát és meghívjuk az
//     Indit() metódusát
        Vezerlo vezerloprog = new Vezerlo();
        vezerloprog.Indit();
// rövidebben, Vezerlo típusú változó létrehozása nélkül
// így is lehet az Indit() metódust meghívni
        new Vezerlo().Indit();
    }
}

```

Befejezésül álljon itt a futási eredmény:

```

A Boszorkány u. 2 -ben lévő lakás bérlöje: Gipsz Jakab
a lakásért havonta 43200 Ft bérleti díjat fizet.

A Rókus u. 2 -ben lévő lakás bérlöje: Gipsz Vilma
a lakásért havonta 38400 Ft bérleti díjat fizet.

Az önkormányzat havi bevétele: 81600 Ft.

```

2.2. Gyakorlati példa (Achs Á.)



Az állatmenhely-alapítvány kisállatversenyet rendez. Mindegyik **állat** regisztrálásakor meg kell adni az állat *nevét* és *születési évét*. Ezek a verseny során nyilván nem változhatnak. Mindegyikötük pontozzák, pontot kapnak a szépségükre és a viselkedésükre is.

A *pontszám* meghatározásakor figyelembe veszik a korukat is (csak év): egy egységesen érvényes *maximális kor* fölött 0 pontot kapnak, alatta pedig az életkor arányában veszik figyelembe a szépségre és a viselkedésre adott pontokat. Minél fiatalabb, annál inkább a szépsége számít, és minél idősebb, annál inkább a viselkedése. (Ha pl. 10 év a maximális kor, akkor egy 2 éves állat pontszáma: (10 – 2) a szépségére adott pontok + 2 a viselkedésre kapott pontok.)

Az állatok adatainak kiíratásához írjuk meg az állat nevét és pontszámát tartalmazó *ToString()* metódust.

Adja meg az aktuális évet és a versenyzők korhatárát (maximális kor), majd kezdje versenyeztetni az állatokat. Ez a következőt jelenti: egy állatnak regisztrálnia kell, majd azonnal kap egy-egy véletlenül generált pontszámot a szépségére is és a viselkedésére is. A pontozás után azonnal írja ki az állat adatait. Mindezt addig ismételje, amíg van versenyző állat. Ha már nincs, akkor írassa ki azt, hogy hány állat versenyzett, mekkora volt az átlag-pontszámuk és mekkora volt a legnagyobb pontszám.

Megoldásjavaslat

A legtöbb program esetén több jó megoldás is lehet. Ezért egyetlen programot se magoljon be, hanem értse meg, csinálja végig a leírtak alapján, majd próbálja meg önállóan is, illetve oldja meg az önálló feladatként megadott részleteket is.

Először is gondolkodjunk el a feladaton, és tervezzük meg a megoldását.

A feladat állatokkal foglalkozik. Ezért célszerű avval kezdeni, hogy definiáljuk az állat fogalmát – legalábbis azt, amit a program során állat fogalmán értünk majd.

Amikor meg akarjuk alkotni az állat fogalmát, akkor (lévén, hogy most program szintjén akarjuk ezt megfogalmazni) egy osztályt definiálunk hozzá. Ez az osztály tartalmazza majd az állat fogalmával kapcsolatos elképzéseinket, azt, hogy milyen adatok és milyen viselkedés (vagyis az adatokkal végzett műveletek) jellemző egy általunk elképzelt állatra. Ez még csak „fogalom”, „tervrajz”, úgy is mondhatjuk, hogy ebben az osztályban írjuk le, azaz itt határozzuk meg az állat fogalmát. Ha ezt megtettük, vagyis definiáltuk az *Allat* nevű osztályt, akkor ettől kezdve deklarálni tudunk *Allat* típusú változókat, és létre tudunk hozni ilyen típusú példányokat.

A feladat szövegének kiemelései segítenek abban, hogy végiggondoljuk, mi minden jellemz egy állatot.

Az állatot jellemző adatok (ezeket **mezőknek** nevezük, néha az adattag elnevezés is használatos): Az állat *neve* és *születési éve* – ezeket azonnal, vagyis már a regisztráció során meg kell adnunk, értékük később nem változhat.

A feladat szerint mindegyikük kap egy *szépségpontot* és egy *viselkedéspontot*. Ezek is mezők. Pontozzák őket – ez már érezhetően valamilyen metódus, azaz az adatokkal végzett művelet. A szöveg megfogalmazásából az is érződik, hogy a pontozás valamilyen külső cselekvés, vagyis `void` metódus tartozik majd hozzá, amelynek során az állat pontszámot kap. Egészen pontosan: két pontszámot kap (a paraméterlistán), és ezek alapján számoljuk ki a versenyben figyelembe vehető tényleges pontszámot. Ennek kapcsán kétféle módon gondolkozhatunk:

a/ Ez a metódus számolja ki a tényleges pontszámot is. Ekkor a pontszám is adattagnak (mezőnek) tekintendő.

b/ Ez a metódus csak a két adott pontszám értékének megadására szolgál, és egy külön `int` típusú metódus számolja ki a pontszám értékét.

Először maradunk az első elképzélésnél, de később kitérünk a másodikra is. Ezek szerint most kell még egy *pontszám* mező is.

Még két további speciális mezőt kell végiggondolnunk. Ki kell számolnunk az állatok életkorát, ezért ehhez meg kell adnunk az *aktuális* évet is. Mivel ez a verseny éve, és ugyanakkor versenyzik az összes állat, ezért ez az évszám nyilván mindenkor esetén azonos, vagyis fölösleges annyiszor létrehozni, ahány állat van, elég, ha csak egyetlen példány létezik belölle. Emiatt ezt majd statikusnak (`static`) deklaráljuk. (De mivel ezt a programot esetleg több évben is használni szeretnénk, ezért nem konstans évszámmal dolgozunk.)

Ugyancsak ez a helyzet a *korhatárral* is, hiszen ez is egyformán vonatkozik minden állatra.

A mezők áttekintése után gondoljuk végig az *Allat* osztályhoz tartozó **metódusokat** is. Ezek: Az állat *korának kiszámítása*, a *pontozás*, illetve a feladat kér még egy speciális metódust, ez az úgynevezett `ToString()`, amely az állatok kiválasztott adatainak `string` formában való megadására szolgál. Erről később majd kicsit részletesebben is lesz szó.

Még egy dologra szükségünk van. Arról volt szó, hogy az állat nevét és születési évét a regisztráció során azonnal meg kell adni, nélküle nem is versenyezhet. (Akár azt is mondhatjuk, hogy a program számára enélkül nem is létezhet egy állatpéldány.)

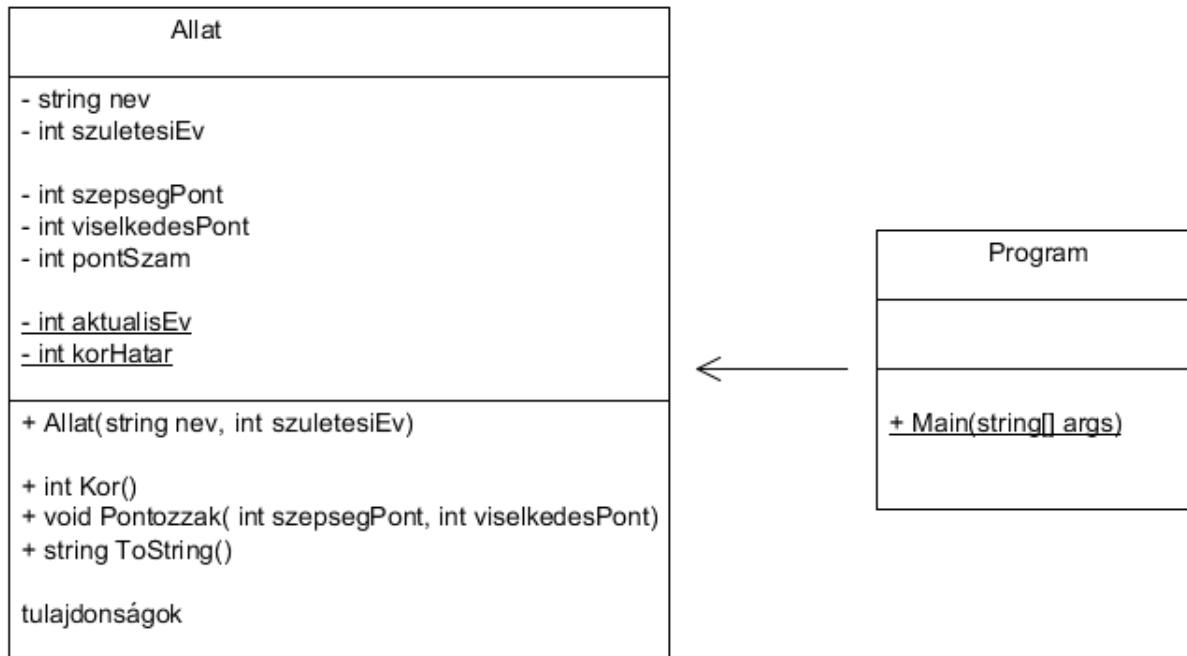
A példányt az úgynevezett **konstruktur** hozza létre (ez konstruálja). A konstruktur a példány létrehozásáért is, de bizonyos mezők inicializálásáért is felelős. Alapértelmezetten minden mező inicializálva van (a típusától függően `0`, `false` vagy `null` az értéke), de nyilván lehetnek olyanok is, amelyeket nem ilyen kezdőértékkel akarunk beállítani. Esetünkben ilyen például a név és a születési év. Ezek a mezők majd a konstruktoron belül kapnak értéket, mégpedig azokat, amelyek a konstruktur paraméterében szerepelnek. Tehát definiálnunk kell majd egy kétparaméteres konstruktort is.

A mezőkkel kapcsolatban még egy fontos dolgot meg kell beszélnünk. Az objektumorientált programozás egyik alapelve az úgynevezett egységezás elve. Ez többek között azt jelenti, hogy a mezőket kizárolag csak az osztályban definiált módon lehet lekérdezni és módosítani. Ezért az összes mezőt mindig private jelzővel látjuk el. Ez azonban azt jelentené, hogy soha senki nem fér hozzá ezekhez az adatokhoz, kizárolag az osztály belügye lenne, hogy melyikkel mit csinál. Nyilván lehetnek olyan mezők is, amelyek értékét bárki lekérdezheti, illetve olyanok is, amelyeket szabad módosítani. Ezekhez azonban külön metódusokat kell írnunk. A C# az ilyen típusú metódusokat speciális módon kezeli, megírni is speciális módon írhatjuk meg őket, sőt külön nevet is kaptak. Ezeket nevezik összefoglaló néven **tulajdonságoknak** (properties).

Ha megalkottuk az állat fogalmát (azaz megírtuk az `Allat` osztályt), akkor ettől kezdve már hivatkozhatunk erre a fogalomra, vagyis definiálhatunk ilyen típusú változókat, létrehozhatunk ilyen típusú példányokat, és dolgozhatunk is velük.

Ezt természetesen egy másik osztályban tesszük, esetünkben most a `Program` osztályban.

Foglaljuk össze az eddigieket egy ábrába (a tulajdonságokat most nem részletezzük):



Jelölések:

- : privát
- + : publikus
- aláhúzás: statikus

Az ábra mutatja az Allat osztály felépítését, és azt is, hogy a Program osztályból fogjuk használni. (Ez az úgynevezett UML ábra – UML: Unified Modeling Language.)

Ezek után írjuk meg az Allat osztályt!

(Ugyanabban a névtérben dolgozunk, amelyben a Program osztály is szerepel, ezért a névteret sehol sem jelezzük.)

Először lássuk az osztály szerkezetét:

```
class Allat {  
  
    // mezők  
    private string nev;  
    private int szuletesiEv;  
  
    private int szepsegPont, viselkedesPont;  
    private int pontSzam;  
  
    private static int aktualisEv;  
    private static int korHatar;  
  
    // konstruktor  
    public Allat(string nev, int szuletesiEv) ...  
  
    // metódusok  
    public int Kor() ...  
  
    public void Pontozzak(int szepsegPont, int viselkedesPont) ...  
  
    public override string ToString() ...  
  
    // tulajdonságok  
  
    // kívülről nem változtatható értékek  
  
    public string Név ...  
    public int SzuletesiEv ...  
    public int SzepsegPont ...  
    public int ViselkedesPont ...  
    public int PontSzam ...  
  
    // kívülről változtatható értékek  
  
    public static int AktualisEv ...  
    public static int KorHatar ...  
}
```

A konstruktur és az első két metódus nem okozhat problémát. A `ToString()` metódusról ejtünk még néhány szót. Ez egy speciális metódus, a metódusfejben lévő `override` jelző azt jelenti, hogy a minden osztály közös ősosztályából, azaz az `Object` osztályból szárma-

zik. Az objektum `string` formáját adja meg, de természetesen azt mi döntjük el, hogy mi legyen ez a `string`. Nyilvánvalóan bármilyen más `string` típusú metódust is írhatnánk, a `ToString()` annyiban speciális, hogy minden típushoz tartozik ilyen metódus, illetve még abban, hogy kiíratáskor nem kell megadnunk a metódus nevét, elég csak az objektumét.

Ezek után nézzük a konstruktort és a metódusokat:

```
// konstruktor
public Allat(string nev, int szuletesiEv) {
    this.nev = nev;
    this.szuletesiEv = szuletesiEv;
}

// metódusok
public int Kor() {
    return aktualisEv - szuletesiEv;
}

public void Pontozzak(int szepsegPont, int viselkedesPont) {
    this.szepsegPont = szepsegPont;
    this.viselkedesPont = viselkedesPont;
    if (Kor() <= korHatar) {
        pontSzam = viselkedesPont * Kor() + szepsegPont * (korHatar - Kor());
    } else {
        pontSzam = 0;
    }
}

public override string ToString() {
    return nev + " pontszáma: " + pontSzam + " pont";
}
```

Már csak a tulajdonságok megbeszélése maradt hátra.

Arról volt szó, hogy a név és a születési év nem változtatható. De kívülről nem változtathatjuk meg a pontszám értékét sem, hiszen az csalás lenne, mert ez a változó kizárolag a `Pontozzak()` metódusban kaphat értéket. Ugyanakkor egyértelmű, hogy bárki tudhatja az állatpéldány nevét, születési évét, megkérdezheti a kapott pontszámát. Vagyis lekérdezni engedjük ezeket az adatokat, de módosítani nem. Ezért ezekhez `get` hozzáférést biztosítunk. Ez azt jelenti, hogy egy speciális `get` nevű metódusban visszaadjuk a változó értékét.

Érdekes kérdés a szépségpont és viselkedéspont helyzete. Ezek nyilván kívülről kapnak értéket, hiszen egy zsűri mondja meg, hogy melyik mekkora, ugyanakkor mégsem írhatunk hozzájuk módosító (vagy beállító, azaz `set` hozzáférést), hiszen a zsűrin kívül senki más nem módosíthatja ezeket a pontokat. Ők viszont a `Pontozzak()` metódusban adják meg ezeket az értékeket.

A tulajdonságokat ugyanúgy szokás nevezni, mint a változókat, csak míg a változó kis kezdőbetűs, addig a tulajdonságneveket nagy kezdőbetűvel írjuk. Ezek alapján a csak lekérdezhető tulajdonságok:

```

// tulajdonságok

// kívülről nem változtatható értékek

public string Nev {
    get { return nev; }
}

public int SzuletesiEv {
    get { return szuletesiEv; }
}

public int SzepsegPont {
    get { return szepsegPont; }
}

public int ViselkedesPont {
    get { return viselkedesPont; }
}

public int PontSzam {
    get { return pontSzam; }
}

```

Kívülről, mégpedig egyszerű értékbeállítással adjuk meg az aktuális év és a korhatár értékét. Az ilyen tulajdonságokhoz set hozzáférést is biztosítani kell. A set (beállítás) azt jelenti, hogy a változó értékét kap, mégpedig azt, amelyet az osztály (illetve objektum) használója ad.

Az, hogy az osztály vagy az objektum használója, attól függ, hogy milyen a tulajdonság. Az olyan mezők, melyek minden példány esetén egyediek, a konkrét objektumon keresztül kapnak majd értéket. Az aktuális év és a korhatár minden egyes példányra egyformán érvényes, ezért csak egyetlen memóriahelyen tároljuk őket, vagyis ezeket osztály szinten adjuk meg. Ezt a tulajdonságok megadása esetén is ugyanúgy jelöljük, mint a mezők esetén, vagyis a static kulcsszóval.

A set beállítás tehát hozzárendeli a változóhoz a felhasználó által adott értéket. Ezt metódusok esetén úgy oldják meg, hogy az átadandó érték a metódus paraméterében szerepel majd. Mint már korábban említettük, a C# speciális metódust használ a mezők értékének megadásához, vagyis itt nem paraméterezzük a metódust, hanem kicsit másképp definiáljuk. Tulajdonságok definiálásakor a mező felveszi majd a speciális value értéket, ami nem más, mint a felhasználó által megadott érték.

```

// kívülről változtatható értékek

public static int AktualisEv {
    get { return aktualisEv; }
    set { aktualisEv = value; }
}

public static int KorHatar {
    get { return korHatar; }
    set { korHatar = value; }
}

```

A set/get hozzáférőket ugyanúgy írjuk meg példányváltozókhoz tartozó tulajdonságok esetén is, mint osztályváltozók (azaz statikus változók) esetén. Jelen feladatban azonban csak a statikus tulajdonságokhoz definiálhatunk set hozzáférést.

Készen vagyunk az `Allat` osztály definíálásával. Nézzük meg, hogyan lehet felhasználni az itt megírt fogalmat.

Erre a `Program` osztályban kerül sor. Először csak egyetlen példányt hozunk létre, és ennek íratjuk ki az eredményét. Bár a későbbiekben megtanuljuk, hogy a `Main()` metódus nem arra szolgál, hogy itt fogalmazzuk meg a programlogikát, hanem csak arra, hogy elindítssük a programot, egyelőre itt, ebben a metódusban hozzuk létre a példányt.

```
class Program {

    static void Main(string[] args) {

        // az allat nevű változó deklarációja
        Allat allat;

        int aktEv = 2015, korhatar = 10;

        string nev;
        int szulEv;
        int szepseg, viselkedes;

        // Az aktuális év és a korhatár megadása
        Allat.AktualisEv = aktEv;
        Allat.KorHatar = korhatar;

        // csak egyetlen példány kipróbálása:
        nev = "Vakarcs";
        szulEv = 2010;
        szepseg = 5;
        viselkedes = 3;

        // az allat példány létrehozása
        allat = new Allat(nev, szulEv);

        // a pontozási metódus meghívása
        allat.Pontozzak(szepseg, viselkedes);

        Console.WriteLine(allat);

        Console.ReadKey();
    }
}
```

Miután egyetlen adattal kipróbáltuk, oldjuk meg az eredeti feladatot is, vagyis tényleg több résztvevője legyen a versenynek. Ezt már nem ilyen csúnyán, ömlesztve írjuk meg, hanem metódus segítségével:

```

    static void Main(string[] args) {
        int aktEv = 2015, korhatar = 10;

        // Az aktuális év és a korhatár megadása
        Allat.AktualisEv = aktEv;
        Allat.KorHatar = korhatar;

        AllatVerseny();

        Console.ReadKey();
    }

```

Továbblépés előtt egyetlen dolgot kell megjegyezni: statikus metódusból csak statikus metódus hívható. Mivel a `Main()` kötelezően statikus, ezért a belőle hívható metódusok is azok. (Ennek kikerüléséről később lesz majd szó.)

A metódus:

```

private static void AllatVerseny();
// az állat nevű változó deklarálása
Allat allat;

string nev;
int szulEv;
int szepseg, viselkedes;
int veletlenPontHatar = 10;

// egy Random példány létrehozása
Random rand = new Random();

// számoláshoz szükséges kezdőértékek beállítása
int osszesVersenyzo = 0;
int osszesPont = 0;
int legtobbPont = 0;

Console.WriteLine("Kezdődik a verseny");

char tovabb = 'i';
while (tovabb == 'i') {
    Console.Write("Az állat neve: ");
    nev = Console.ReadLine();

    Console.Write("születési éve: ");
    while (!int.TryParse(Console.ReadLine(), out szulEv)
        || szulEv <= 0
        || szulEv > Allat.AktualisEv) {
        Console.Write("Hibás adat, kérem újra.");
    }

    // véletlen pontértékek
    szepseg = rand.Next(veletlenPontHatar + 1);
    viselkedes = rand.Next(veletlenPontHatar + 1);
}

```

```

// az állat példány létrehozása
allat = new Allat(nev, szulEv);

// a pontozási metódus meghívása
allat.Pontozzak(szepseg, viselkedes);

Console.WriteLine(allat);

//számítások

osszesVersenyzo++;
osszesPont += allat.PontSzam;
if (legtobbPont < allat.PontSzam) {
    legtobbPont = allat.PontSzam;
}

Console.Write("Van még állat? (i/n) ");

// alakítsa át ellenőrzött beolvasásra
tovabb = char.Parse(Console.ReadLine());
}

// eredmény kiíratása
Console.WriteLine("\nÖsszesen " + osszesVersenyzo + " versenyző volt,"
+ "\nösszpontszámuk: " + osszesPont + " pont,"
+ "\nlegnagyobb pontszám: " + legtobbPont);
}
}

```

Megjegyzések:

1. Az eredmény formátum string segítségével is kiíratható, például ehhez hasonló módon:

```

Console.WriteLine("\nÖsszesen {0} versenyző volt, átlaguk {1:###.##} pont.",
osszesVersenyzo, 1.0 * osszesPont / osszesVersenyzo);

```

Az átlag kiszámolásakor használt 1.0 szorzó azért kell, hogy double legyen az eredmény, mert különben két int típusú változó hányadosa is int típusú lenne. Az is fontos, hogy a szorzás megelőzze az osztást, mert az azonos precedenciájú műveleteket balról jobbra haladva értékeli ki a számítógép. Most a szorzás eredménye double, így a hányados is az lesz.

2. A megoldás ellentmond a bevezetőben (1.1. fejezet) említett modularitás, sőt az adat-reprezentáció rugalmassága elvének is, azok szerint ugyanis külön kellene választani a beolvasást a számolástól, vagyis az egyetlen AllatVerseny() metódus helyett három metódust kellett volna írnunk: Beolvasas(), OsszPontSzamitas(), MaximumKereses(). Ehhez azonban az kellene, hogy tároljuk a beolvasott adatokat, azaz vagy tömbbe vagy listába tegyük őket. Ha elég érdeklődő, akkor már most megpróbálhatja így megoldani, egyébként pedig majd a listák tárgyalása után kerül sorra a korrekt megoldás.

3. A megoldás megtervezése során szó volt arról, hogy a pontszám kiszámítását külön metódusba is írhatnánk. Ekkor az `Allat` osztályban nem kell (nem szabad) definiálnunk a `PontSzam` változót és a hozzá tartozó tulajdonságot sem, a megfelelő metódusok pedig így alakulnának:

```
public void Pontozzak(int szepsegPont, int viselkedesPont) {
    this.szepsegPont = szepsegPont;
    this.viselkedesPont = viselkedesPont;
}

public int PontSzam() {
    if (Kor() <= korHatar) {
        return viselkedesPont * Kor() + szepsegPont * (korHatar - Kor());
    }
    return 0;
}

public override string ToString() {
    return nev + " pontszáma: " + PontSzam() + " pont";
}
```

Ennek megfelelően természetesen a felhasználás során is megváltozik a pontszámok használata, mégpedig így:

```
osszesPont += allat.PontSzam();
if (legtobbPont < allat.PontSzam()) {
    legtobbPont = allat.PontSzam();
}
```

Hurrá! Akkor biztosan van még más jó megoldás is, hiszen eleve úgy indult ez a fejezet, hogy több jó megoldás lehetséges. Ez igaz, de mint kiderül, nem is olyan sok. A most tárgyalt ötlet például jó. De vajon jó-e az az, egyébként akár működőképes megoldás, hogy minden pontszámfajtát mezőként deklaráljuk, a `Pontozzak()` metódus helyett a szépségponthoz és a viselkedésponthoz nemcsak `get`, hanem `set` hozzáférést is biztosítunk a tulajdonság definiálásakor, és csak a most tárgyalt `PontSzam()` metódust írjuk meg?

Ez első olvasásra jónak tűnhet, de ha alaposabban belegondolunk, akkor mégsem az. Miért?

A feladat szerint a zsűri egyszerre kétféle pontot ad. A most említett megoldás lehetőséget adna rá, hogy valamelyik pontozást elsinkófáljuk, vagy esetleg utólag, korrupt módon megváltoztassuk a zsűri döntését. Az eredeti megoldásban tárgyalta paramétereit erre nem ad lehetőséget.

Önálló továbbfejlesztésként egészítse ki az eddig tárgyaltakat úgy, hogy az állat regisztrálásakor meg kelljen adni az oltási igazolásának számát (később nem változhat), illetve a verseny kezdetekor (vagyis már csak a regisztrálás után) mindegyikük kapjon egy-egy rajtszámot.

Módosítsa a `ToString()` metódust is, és vegye bele az állat rajtszámát is.

A vezérlésben a regisztrálás sorrendjében kapják meg ezt a rajtszámot.

3. A C# listakezelése

3.1. Elméleti háttér (Szendrői E.)

A második fejezet elméleti részében bemutatott program, amely a lakásbérleti díját számolja ki, két tesztadattal működött. Az önkormányzat azonban sok lakással rendelkezik, tehát több lakást akar bérbe adni. A bérbe adandó lakáspéldányokat tárolni kellene valamilyen alkalmas formában. Kézenfekvő gondolat, hogy tároljuk a lakáspéldányokat olyan szerkezetben, amely lehetővé teszi, hogy újabb példányokat adjunk a korábbiakhoz, rendezni tudjuk, könnyen el tudjuk érni bármelyik elemet stb. Egyik lehetőség, hogy a lakáspéldányokat tömbben tároljuk, de ez nem túl rugalmas megoldás. Kézenfekvőbb, ha olyan szerkezetet választunk, amelyben dinamikusan tudjuk növelni vagy csökkenteni az elemek számát. Mindkét lehetőséget tekintsük át.

3.1.1. Tömbök kezelése

A tömbök a System.Array osztályból származnak. Egy tömb minden eleme azonos típusú. A tömb referenciátípus, elemei a memóriában folytonos blokkokban tárolódnak. A tömbök elemeire indexeléssel hivatkozhatunk, az elemek sorszámozása 0-tól indul. A tömbök lehetnek egydimenziós (vektor), kétdimenziós (mátrix) és többdimenziós tömbök, valamint jagged (egyenleten vagy feldarabolt) tömbök. A tömb **Rank** (rang) tulajdonsága megadja a tömb dimenzióinak számát. A tömb utolsó érvényes indexét megkapjuk a *GetUpperBound()* metódus hívásával. A metódus paramtere a dimenzió sorszáma, amely hasonlóan az indexhez, 0 kezdőértékről indul. Tömböket létrehozhatunk *struktúrákból*, *felsorolásokból* (*enum*) és *osztályokból* is.

Egydimenziós tömbök

A egydimenziós tömböket a [] operátorral hozzuk létre, melynek formája a következő:

```
típus [ ] tömbnév; // még nincs memória foglalás
```

Példányosítás a new kulcsszóval (memória foglalás) történik. A tömb deklarálásakor megadhatjuk az elemek értékét, ezzel egyúttal a tömb méretét is megadjuk. A méret megadása inicializálás nélkül a new utasítást követően a [] zárójelbe beírt értékkel történik.

```
int [] nums= new int [] {5, -1, 12, 8, 5} //azonnal inicializáljuk  
double[] num2= new double[5];
```

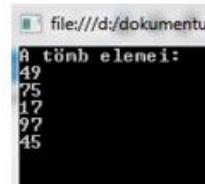
A tömb deklarációja után, ha nem inicializáltuk deklaráláskor a tömb elemeit, az egyes elemek automatikusan a tömb típusának megfelelő null értékre inicializálódnak.

Hasznos tulajdonsága a **Length**, amely az elemek számát adja meg. Ennek megfelelően a legutolsó elem indexe az elemek száma-1 (Length-1).

A fordító nem ellenőrzi fordítási időben az indexek helyességét. Ezért hibás indexelés esetén futásidőben `System.IndexOutOfRangeException` kivételt fog dobni a program. A kivételkezeléssel később foglalkozunk.

Példa egydimenziós tömb használatára, amelyben egy egész típusú 5 elemű vektort feltöltünk véletlen számokkal, majd kiíratjuk az elemeket a képernyőre.

```
static void Main(string[] args)
{
    int n=5;
    Random r=new Random();
    int[] vektor;
    vektor = new int[n];
    // feltöljtük a vektort 100-nál kisebb véletlen számmal
    for (int i = 0; i < vektor.Length; i++)
    {
        vektor[i] = r.Next(100);
    }
    // kiíratjuk a vektor elemeit foreach ciklussal
    Console.WriteLine("A tömb elemei:");
    foreach (int elem in vektor)
    {
        Console.WriteLine(elem);
    }
    Console.ReadKey();
}
```



Többdimenziós tömbök

A többdimenziós tömbök közül leggyakrabban a kétdimenziós tömböket vagy mátrixokat használjuk. A C# nyelvben maximum 32 dimenziós tömböt deklarálhatunk.

Kétdimenziós, egész típusú tömb deklarálása a következő módon történik:

```
int[,] tabla=new int[4,6];
```

A deklarálás történhet az elemeinek felsorolásával (3×3 -as mátrix) is.

```
int[,] matrix = new int[,]
{
    {12, 53, 2},
    {13, 76, 52},
    {45, 25, 3}
};
```

A kétdimenziós tömb elemeire a sor- és oszlopindex megadásával hivatkozunk. Az elemek kezelése egymásba ágyazott ciklusokkal történik. A `Length` tulajdonság, ahogy fent említettük, a tömb elemének a számát adja meg, tehát a fenti 3×3 -as tömb esetén a `Length` tulajdonság értéke 9 lesz.

Lássunk egy példát kétdimenziós tömb kezelésére! A példában 3×3 -as egész típusú tömböt töltünk fel véletlen számokkal, majd kiíratjuk a tömb elemeit a képernyőre. Látható, hogy a

ciklusváltozók felső határának megadásához a tömb adott dimenzióból elemszámot visszaadó GetLength() metódust hívtuk meg. A metódus paramétere a dimenziószám. A sor a 0. dimenzió, az oszlop az 1. dimenzió.

```
static void Main(string[] args)
{
    int[,] matrix = new int[3, 3];
    Random r = new Random();
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = r.Next();
        }
    }
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            Console.WriteLine(matrix[i, j]);
        }
        Console.WriteLine("\n");
    }
    Console.ReadKey();
}
```

Jagged tömbök

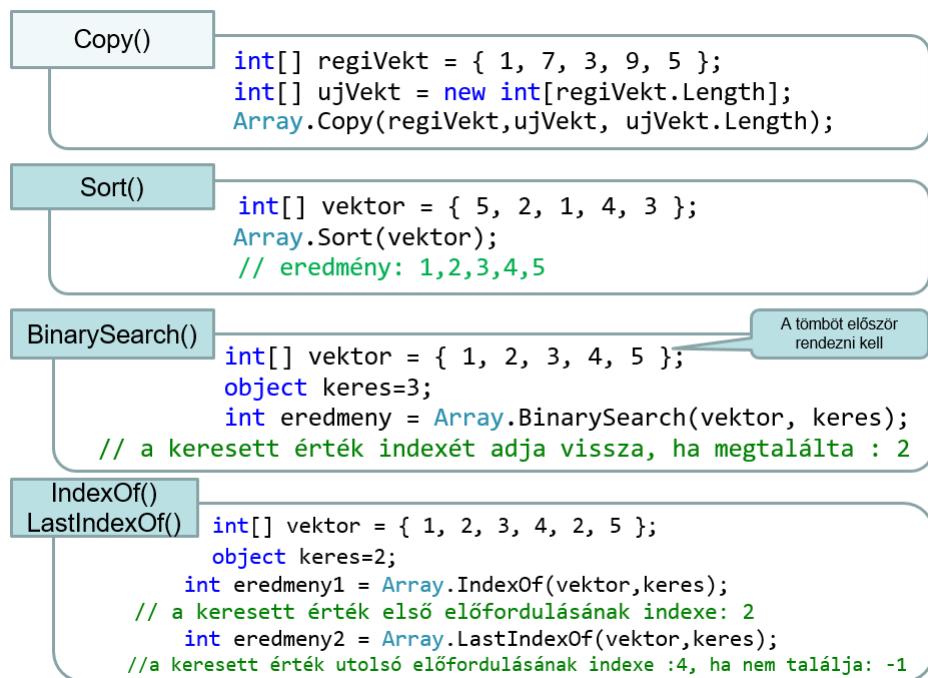
A többdimenziós tömbök egyik variánsa az ún. egyenetlen (jagged) (feldarabolt) tömb. Ekkor legalább egy dimenzió hosszát meg kell adnunk, ez konstans marad, viszont a belső tömbök hossza tetszés szerint megadható. A következő példában egy 3 soros, soronként változó elemszámú tömböt hozunk létre.

```
int[][] jagtomb = new int[3][];
```

A System.Array osztály néhány hasznos metódusa

Gyakori feladat, hogy tömbök elemeit másik tömbbe kell átmásolni, vagy keresni kell valamilyen értéket a tömbben, vagy éppen rendezni szeretnénk a tömb elemeit. Ezeknek a feladatoknak az elvégzése saját kód írása nélkül is megoldható, ha igénybe vesszük a System.Array osztály statikus és példánymetódusait. A *CopyTo()* egy adott kezdő indextől indulva egy tömb elemeit átmásolja egy másik tömbbe. Egyik tömbből a másikba értékeket a *Copy()* statikus metódussal is másolhatunk. Paraméterként a forrás, a cél tömböt, valamint a tömbméretet kell megadni. További lehetőség a *Clone()* példánymetódus használata, melynek a segítségével egy lépésben hozhatunk létre és másolhatunk át egy teljes tömböt.

A másoláson túl a tömbök elemeinek rendezésére a *Sort()* metódus, keresésre a *BinarySearch()*, *IndexOf()* és *LastIndexOf()* metódusokat használhatjuk. A következő ábra bemutatja ezeknek a metódusoknak a használatát.



12. ábra A System.Array osztály néhány statikus metódusa

A második fejezetben megbeszélt lakásbérbeadás bevételét számoló alkalmazást módosítottuk, már nemcsak két lakással működik a program, hanem tömb bevezetésével az általunk megadott számú lakást képes kezelni. A **Lakas** osztály nem változott. A **Vezerlo** osztályban kellett módosítani a kódot. Bevezettünk egy **Lakas** típusú egydimenziós tömböt, ebben tároljuk a lakás objektumokat. A módosított kód alább tekinthető meg.

```

public class Vezerlo
{
    private const int n = 5;
    private Lakas[] lakkasok=new Lakas[n];

    public void Indit() {
        StatikusokBeallitasa();
        Beolvashas();
        Console.WriteLine("\n A bérbe adható lakások:\n");
        Kiiratas();
        Berbeadas();

        Console.WriteLine("\nAz önkormányzat havi bevételei:\n");
        Bevetelek();
        Console.ReadKey();
    }
}

```

Objektumtömb

A következő programrészlet a *Beolvasas()* metódus utasításait tartalmazza. Itt hozzuk létre ciklusban a lakás objektumpéldányokat és helyezzük el a létrejött objektumot a tömbben.

```
private void Beolvasas() {
    string helyrajziSzam, komfortfokozat, cim;
    int terulet;
    for (int i = 0; i < n; i++)
    {
        Console.WriteLine("\nKérem a {0}. lakás helyrajziszámát: ", i+1);
        helyrajziSzam = Console.ReadLine();
        Console.WriteLine("Kérem a lakás címét: ");
        cim = Console.ReadLine();
        Console.WriteLine("Kérem a lakás alapterületét: ");
        terulet = int.Parse(Console.ReadLine());
        Console.WriteLine("Kérem a lakás komfortfokozatát: ");
        komfortfokozat = Console.ReadLine();
        lakasok[i] = new Lakas(helyrajziSzam, cim, terulet, komfortfokozat);
    }
}
```

A tömb feltöltése

A tömbben tárolt objektumok kiíratását a *Kiiratas()* metódus végzi.

```
private void Kiiratas() {
    foreach (Lakas lakas in lakasok)
    {
        Console.WriteLine("Helyrajziszám: " + lakas.HelyrajziSzam + " Cím: "
            + lakas.Cim + "\n\talapterület: " +
            " komfortfokozat: " + lakas.Komfortfokozat);
        Console.WriteLine();
    }
}
```

A *Berbeadas()* metódus a tulajdonos nevét kéri be. A *Bevetelek()* metódus kiszámolja az önkormányzat bevételét. Egy foreach ciklus segítségével halad végig a tömb elemein és számolja a bérleti díjat és az önkormányzat bevételét.

```
private void Berbeadas() {
    Console.WriteLine("Lakások bérbe adása");
    for (int i = 0; i < n; i++)
    {
        Console.Write("\nKérem a {0} lakás bérlöjének nevét: ", i + 1);
        lakasok[i].BerloNeve = Console.ReadLine();
    }
}

1 reference
private void Bevetelek() {
    int ossz = 0;
    foreach (Lakas lakas in lakasok)
    {
        Console.WriteLine(lakas);
        ossz += lakas.HaviBerletiDij();
    }
    Console.WriteLine("\nAz önkormányzat havi bevétele összesen " + ossz + " Ft.");
}
```

3.1.2. Gyűjtemények, listák kezelése

A tömbök használatának megvannak a korlátai. Egy fontos hátránya a tömböknek, hogy nem növelhető és nem is csökkenthető futás közben az elemek száma, hiszen a tömb méretét már a deklaráció során meg kell adnunk. Nem tudunk eltávolítani elemet a tömbből, és nem tudunk új elemeket beszúrni. Ezeket a problémákat a *System.Collections* névtér és alnévterében található **Collection** (gyűjtemény) osztályok használatával orvosolhatjuk. A gyűjteményosztályok objektumként fogadják, tárolják és adják vissza az elemeket, vagyis az elemek típusa *object*.

Az ArrayList osztály

Dinamikus tömbkezelést tesz lehetővé, a tömbelemek száma rugalmasan módosítható. Hasonlóan a hagyományos tömbökhöz, a tömblista elemeire indexeléssel hivatkozunk. Az indexelés zéró-bázisú, azaz 0-val kezdődik. A *Capacity* tulajdonsággal a helyfoglalás lekérdezésére nyílik lehetőség. Az *ArrayList* osztály számos hasznos metódussal rendelkezik, amellyel megkönnyítik a tömbök dinamikus kezelését. Ezek a metódusok a következők:

- *Add()* – Egy *ArrayList* osztály végéhez új elemet fűzhetünk hozzá
- *Insert()* – *ArrayList* elemei közé tehetünk új elemeket.
- *Clear()* – Törli az elemeket
- *RemoveAt()* – Eltávolít egy adott pozíciójú elemet
- *Remove()* – Adott objektumelemet eltávolít
- *Sort()* – Rendezi az elemeket
- *Reverse()* – Fordított sorba rendez
- *BinarySearch()* – Gyors keresést tesz lehetővé.

Egyéb más gyűjtemények kezelését is támogatja a C# programozási nyelv, ezekre nem térünk ki részletesen, csak felsoroljuk őket. Ezek a következők: a Queue osztály, Stack osztály, Hashtable osztály, SortedList osztály.

Listák

A lista (*List<Típus>*) egy olyan gyűjtemény, amelyhez elemeket adhatunk hozzá (*Add()* metódus), amelybe elemeket lehet beszúrni egy adott indexű helyre, ezeket az elemeket az index alapján el lehet érni, keresni lehet benne, illetve törölni. A listáknak van néhány fontos tulajdonsága és metódusa. Kényelmesebben használhatók, mint a tömbök.

Néhány fontos metódus az *Add()*, az *Insert()*, amelyek segítségével új elemet adhatunk a listához. A *Remove()*, *RemoveAt()* metódussal elemet távolíthatunk el, a *Clear()* metódussal

törölhetjük a teljes listát. Legfontosabb tulajdonsága a Count, amely megadja, hogy aktuálisan hány elem van a listában.

A lista deklarációja, feltöltése, jelenleg lakás objektumokkal, az alábbi kódrészleten látható:

```
private List <lakasok>= new List<Lakas>();  
  
lakas = new Lakas(hrsz, cim, terulet, komfortfokozat);  
lakasok.Add(lakas);
```

A következő egyszerű példán bemutatjuk a listák használatát. Feltöltünk egy listát néhány névvel, majd kiírjuk a lista elemeinek számát a feltöltést megelőzően, majd azt követően. Ehhez a Count tulajdonságot használjuk fel.

```
namespace ListakDemo1  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // string típusú lista objektum létrehozása  
            List<string> nevLista = new List<string>();  
  
            // a lista elemeinek száma  
            Console.WriteLine(" A névlista elemeinek száma: " + nevLista.Count);  
  
            // elemek hozzáadása a string listához  
            nevLista.Add("Ágnes");  
            nevLista.Add("Kati");  
            nevLista.Add("János");  
            nevLista.Add("Zsuzsa");  
            nevLista.Add("Péter");  
            // a lista elemeinek száma  
            Console.WriteLine("\n A névlista elemeinek száma: " + nevLista.Count+  
                " a feltöltés után\n");  
        }  
    }  
}
```

A kiíratás eredménye:

```
 A névlista elemeinek száma: 0  
 A névlista elemeinek száma: 5 a feltöltés után
```

A következő kódrészletben kétféle módon is kiírjuk a lista elemeit. Először *for* ciklusban, felhasználva a Count tulajdonságot, a ciklusváltozó felső határának megadásához. A második esetben külön metódusban *foreach* ciklusban történik meg az elemek kiíratása. Közben arra is látunk példát, hogyan tudunk elemet eltávolítani a listából a RemoveAt() metódussal, melynek paramétereként meg kell adni az eltávolítandó elem indexét.

```

// a lista elemeinek kiíratása
for (int i = 0; i < nevLista.Count; i++)
{
    Console.WriteLine("A lista " + i + "-dik eleme: " + nevLista[i]);
}
// Elemek eltávolítása a listából
// Az 2-es indexű (harmadik) elem eltávolítása
nevLista.RemoveAt(2);
Console.WriteLine("\nA 2-es indexű elem eltávolítása után");
ListaKiiratas(nevLista);

// Véti eltávolításra a listából

```

Count tulajdonság, a lista elemeinek számát adja meg

A lista i-dik eleme

3 references

```

private static void ListaKiiratas(List<string> nevLista)
{
    Console.WriteLine("\n A lista elemei\n");
    foreach (var item in nevLista)
    {
        Console.WriteLine(item);
    }
}

```

A lista 0-dik eleme: Ágnes
 A lista 1-dik eleme: Kati
 A lista 2-dik eleme: János
 A lista 3-dik eleme: Zsuzsa
 A lista 4-dik eleme: Péter
 A 2-es indexű elem eltávolítása után
 A lista elemei
 Ágnes
 Kati
 Zsuzsa
 Péter

Az alábbi utasítások a Remove() metódus működését mutatják be. Paramétere a törlendő elem értéke. A Remove() metódus visszatérési értéke logikai, és azt jelzi, hogy sikeres volt-e a törlés művelete vagy nem.

```

// Kati eltávolítása a listából
// A Remove() metódus logikai értékkel tér vissza
if(!nevLista.Remove("Kati"))
{
    Console.WriteLine("Nincs ilyen név a listában!");
    Console.WriteLine("\nA törlés után megmaradt elemek");
    ListaKiiratas(nevLista);

    // Így elem hozzáírása /hihatalomzás/
}

```

Remove() metódus használata

A 2-es indexű elem eltávolítása után
 A lista elemei
 Ágnes
 Kati
 Zsuzsa
 Péter
 A törlés után megmaradt elemek
 A lista elemei
 Ágnes
 Zsuzsa
 Péter

Az elemek beszúrása a listába az *Insert()* metódussal történik. A metódus első paramétere az index, vagyis meg kell mondanunk, hogy hova akarjuk beszúrni az új elemet. A második paraméter a beszúrandó elem értéke. (Ezzel szemben az *Add()* metódus mindenkor a lista végéhez adja hozzá az új elemet.)

Insert (hova, mit)

```
// Új elem beszúrása (hibajelzést kapunk, ha
// az index értéke kisebb mint 0 vagy nagyobb a lista
// elemeinek számánál (Count) tulajdonság)

nevLista.Insert(0, "Gábor");

Console.WriteLine("Új elem beszúrása után");
ListaKiaratas(nevLista);

// Keresés a listában: Zsuzsát keressük
```

```
Új elem beszúrása után
A lista elemei
Gábor
Ágnes
Zsuzsa
Péter
```

A listában megkereshetjük egy elem indexét, ebben segít az *IndexOf()* metódus, melynek paramétere a listaelem. A lista kiürítése a *Clear()* metódussal történik. Erre látunk példát a következő kódrészletben.

```
// Keresés a listában: Zsuzsát keressük
int pozicio = nevLista.IndexOf("Zsuzsa");

if (pozicio != -1)
{
    Console.WriteLine("\nA keresett elem {0} az {1}-es indexű elem a listában",
                    nevLista[pozicio], pozicio.ToString());
}
else
{
    Console.WriteLine("\nA keresett elem nincs a listában!");
}

// A lista kiürítése
nevLista.Clear();

// a lista elemeinek száma
Console.WriteLine("\n A névlista elemeinek száma: " + nevLista.Count);

Console.ReadKey();
}
```

```
A keresett elem Zsuzsa az 2-es indexű elem a listában
A névlista elemeinek száma: 0
-
```

3.1.3. Listák feltöltése fájlból, fájlba írás

Amikor sok adatot kell megadnunk egy program futtatásakor, például tömböt vagy listát kell feltöltenünk elemekkel, sokkal kényelmesebb az adatok beolvasása egy előre elkészített fájlból, mint adatok sokaságának begépelése.

Minden input és output művelet a .NET-ben adatfolyamok (stream) használatával történik. A stream egy absztrakt reprezentációja a soros eszközöknek (hálózati csatorna, lemez stb.). A fájlok kezelését támogató osztályok a System.IO névterben vannak.

A FileStream osztály byte-okon és byte-tömbökön végzi a műveleteket, míg a **Stream** osztályok szöveges adatokkal dolgoznak. A szöveges fájlok olvasása a StreamReader, írása a StreamWriter osztályok metódusaival történik. A StreamReader osztály egy TextReader, míg a StreamWriter osztály egy TextWriter nevű absztrakt osztályból származik.

A StreamReader vagy StreamWriter objektumok létrehozása a következőképpen történik: Nevezzük ezeket az objektumokat író-, illetve olvasócsatornának.

```
StreamWriter irocsatorna = new StreamWriter(fájlnév);
```

Mivel a konstruktorban megadjuk a fájlnévet, rögtön meg is nyitja. Íráskor, ha létezik már ilyen néven fájl, felülírja. Ha nem adunk meg a fájl nevénél elérési utat, akkor a deafult könyvtárban, a projekt bin\Debug könyvtárában keresi a fájlt.

```
StreamReader olvasocsatorna = new StreamReader(fájlnév);
```

A fájlműveletek lépései:

- A fájlművelet megkezdése előtt létre kell hozni és meg kell nyitni a fájlt.
- Ki- vagy bemenő adatfolyam hozzárendelése a fájlhoz, azaz az író- vagy olvasócsatorna létrehozása a fájl eléréséhez.
- Ezután megtörténhet a fájl írása vagy olvasása.
- Az adatfolyam, illetve fájl lezárása.

Fájlok írása, olvasása a ReadLine(), illetve WriteLine() metódusokkal történik.

```
ReadLine() metódus, példa : stringváltozó = olvasocsatorna.ReadLine();
```

Ha a fájl üres vagy elérte a fájl végét, a ReadLine() metódus *null* értéket ad vissza.

```
WriteLine() metódus, példa: irocsatorna.WriteLine(string);
```

Kapcsolat lezárása a Close() metódussal történik:

```
olvasocsatorna.Close();
```

Fontos, hogy a program **using** direktívai közé vegyük be a **using System.IO** névterre való hivatkozást. Hiányában nem tudunk fájlműveleteket végezni.

Nézzünk egy példát a fájlok kezelésére:

A Beolvasas() metódus egy egész számokat tartalmazó listával tér vissza. A lista adatait a **szamok.txt** fájlból olvassa. Az **olvasocsatorna** objektum létrehozása után rögtön meg is nyílik a fájlkapcsolat, mivel a konstruktorban szerepelt a fájl neve. Ezután egy elől tesztelő ciklusban addig, amíg a fájl végére nem érünk, beolvassunk egy-egy sort a fájlból a ReadLine() metódussal. Konvertálás után (Parse() metódus) a beolvastott számot hozzáadjuk a listához. A ciklus befejeződése után lezárjuk a kapcsolatot. Az EndOfStream logikai típusú tulajdonság, a fájl végét jelzi.

```
private static List<int> Beolvasas()
{
    List<int> szamok=new List<int>();
    int szam;

    // Fájlból beolvasás, majd a listához adás

    StreamReader olvasocsatorna = new StreamReader("szamok.txt");

    while (!olvasocsatorna.EndOfStream)
    {
        szam = int.Parse(olvasocsatorna.ReadLine());
        szamok.Add(szam);
        // vagy
        // szamok.Add(int.Parse(olvas.ReadLine()));
    }
    olvasocsatorna.Close();
    return szamok;
}
```

Példa egész értékeket tartalmazó lista elemeinek fájlba írására, Kiir() metódus. A **StreamWriter** osztály konstruktorával létrehozzuk az **irocsatorna** objektumot, megnyitjuk a fájlt, majd egy *foreach* ciklusban végighaladunk a listán és minden elemét beírjuk a fájlba. Ezt követően lezárjuk a kapcsolatot.

```
// Fájlba írás
0 references
private static void Kiir(List<int>szamok){
    StreamWriter irocsatorna = new StreamWriter("szamlista.txt");
    foreach (var elem in szamok)
    {
        irocsatorna.WriteLine(elem);
    }
    irocsatorna.Close();
}
```

3.2. Gyakorlati példa (Achs Á.)

Divatja van a különféle tévés vetélkedőknek (A dal, X-faktor és társai). De muszáj minden előtt ülni?

Szervezzünk meg egy házi dalversenyt! A versenyen bármelyik szak diákjai indulhatnak, produkciójukat egy zsűri értékeli majd.

A versenyre jelentkező énekeseknek meg kell adniuk a nevüket és a szakjukat, és regisztráláskor mindegyikük kap egy egyedi rajtszámot is. A verseny során egy zsűri tagjai pontozzák őket, minden egyes zsűritag hatására a versenyzők pontszáma a metódus paraméterében megadott értékkel növekszik.



A verseny adminisztrálását segítő program a következőket tudja:

Olvassuk be a versenyzők adatait egy adatfájlból. Fájlszerkezet: soronként: a diák neve, következő sorban a szakja, majd a következő sorban az újabb diák neve stb. Rajtszámuk a beolvasás sorszáma legyen. (A megoldás a *versenyzok.txt* fájlnevet használja majd.)

Már a verseny megkezdése előtt láthatunk a konzolon az adataikat, azaz a versenyző rajtszámát, nevét, szakját és pontszámát. (Így is ellenőrizhető, hogy nincs korrupció, vagyis valóban nulláról indul-e mindenki.)

A verseny most annyiból áll, hogy minden egyes versenyzőt pontoz a zsűri, azaz annyi véletlen pontszámot kap, ahány zsűritag van (a zsűritagok számát és az adható maximális pontszámot konstansként megadhatjuk a program elején).

Természetesen a verseny végén is kíváncsiak vagyunk a kijelzőre.

Állapítsuk meg az eredményeket, vagyis azt, hogy

- Kik kapták a legtöbb pontszámot.
- Mi a verseny végeredménye – azaz rendezzük pontszám szerint csökkenő sorrendbe a versenyzőket.
- Ki lehessen keresni a begépelt szakhoz tartozó versenyzőket. Arra is adjon lehetőséget, hogy a keresést meg lehessen ismételni. Természetesen azt is írja ki, ha senkit sem talál.

Megoldásjavaslat

Gondolkodjunk el, és tervezzük meg a megoldást.

Feladatunk most versenyzők adminisztrálása, ezért először definiáljuk a versenyző általunk használni kívánt fogalmát, azaz megírjuk a `Versenyo` osztályt.

Ez elég egyszerű lesz, hiszen a feladat elsődleges célja most a vezérlés megtárgyalása.

Ahogy látjuk, egy versenyzőt a *rajtszáma*val, *nevével*, *szakjával* és *pontszámával* jellemzhetünk. Vagyis ezek lesznek az osztály mezői. Regisztrációkor meg kell adni a nevét és szakját, és azonnal kap egy rajtszámot is. Vagyis ezek nélkül az adatok nélkül nem is lehet adminisztrálni őt, azaz létre sem jöhet egy `Versenyo` típusú példány. Ez azt jelenti, hogy azonnal meg kell adni a kezdőértéküket, ezért a `rajtSzam`, `nev`, `szak` mezőknek a konstruktorkban kell értéket adnunk. Mivel ezek az értékek később sem változhatnak, ezért nem írhatunk hozzájuk set hozzáférést.

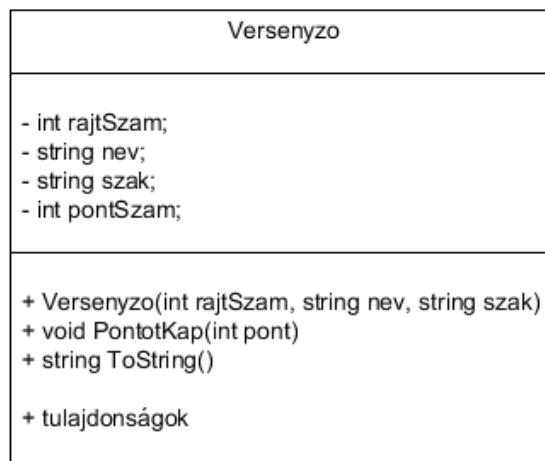
Ugyancsak nem írhatunk set hozzáférést a `pontSzam` mezőhöz sem, mégpedig azért, mert ennek a mezőnek az értéke kizárolag úgy növekedhet, ha egy zsűritag pontozza a versenyzőt. Ezt írjuk meg a `PontotKap()` metódusban.

Mivel egy kijelzőn is szeretnénk megjelentetni a versenyzők adatait, ezért célszerű megírni a `ToString()` metódust is.

Az elmondottakat összefoglaljuk egy-egy táblázatban, majd egy UML diagramon is.

mezőnév	típus	értékkadás	hozzáférés	statikus
<code>rajtSzam</code>	<code>int</code>	konstruktorkban	<code>get</code>	nem
<code>nev</code>	<code>string</code>	konstruktorkban	<code>get</code>	nem
<code>szak</code>	<code>string</code>	konstruktorkban	<code>get</code>	nem
<code>pontSzam</code>	<code>int</code>	metódusban	<code>get</code>	nem

metódusnév	típus	paraméterek	statikus
<code>PontotKap()</code>	<code>void</code>	<code>int</code>	nem
<code>ToString()</code>	<code>string</code>	nincs	nem



Az osztály kódja:

```
class Versenyo {  
  
    // mezők  
    private int rajtSzam;  
    private string nev;  
    private string szak;  
  
    private int pontSzam;  
  
    // konstruktor  
    public Versenyo(int rajtSzam, string nev, string szak) {  
        this.rajtSzam = rajtSzam;  
        this.nev = nev;  
        this.szak = szak;  
    }  
  
    // metódusok  
    public void PontotKap(int pont) {  
        pontSzam += pont;  
    }  
  
    public override string ToString() {  
        return rajtSzam + "\t" + nev + "\t" + szak + "\t" + pontSzam + " pont";  
    }  
  
    // tulajdonságok  
    public int RajtSzam {  
        get { return rajtSzam; }  
    }  
  
    public string Nev {  
        get { return nev; }  
    }  
  
    public string Szak {  
        get { return szak; }  
    }  
  
    public int PontSzam {  
        get { return pontSzam; }  
    }  
}
```

Beszéljük meg az előzőeknél több munkát igénylő vezérlést. Erre, mint gondolnánk, a Program osztályban kerül sor. Ez lényegében így is van, de mégis az eddigiek től eltérő módon oldjuk meg. Az ugyanis egyértelmű, hogy a feladat kellően nagy ahhoz, hogy metódusokra bontva oldjuk meg. Igen ám, de mivel a Main() metódus statikus, ezért az összes, általa hívott metódusnak is statikusnak kell lennie. A statikus metódusok rontják a program rugalmasságát, ezért sokszor (bár nem minden) célszerű kikerülni a használatukat. Jelenleg is ezt tesszük, mégpedig úgy, hogy a vezérlési funkciók számára egy új osztályt hozunk létre, és a Main() metódusban ennek az osztálynak az indító metódusát hívjuk meg. Legyen az osztály neve pl. VezerloOsztaly, a meghívott metódusé pedig Start().

Mivel éppen a statikus metódusokat akarjuk kikerülni, ezért nyilván ez a bizonyos indító metódus sem lehet statikus. Ez viszont azt jelenti, hogy nem az osztályon, hanem egy példányon keresztül lehet elérni, vagyis a metódus hívásához előbb példányosítanunk kell a VezerloOsztaly-t.

A kód:

```
class Program {
    static void Main(string[] args) {
        new VezerloOsztaly().Start();
        Console.ReadKey();
    }
}
```

Mivel a létrehozott VezerloOsztaly típusú példányra a metódus indításán kívül nincs szükségünk, ezért nem is raktuk be a példányt egy változóba, hanem azonnal meghívtuk a metódusát.

Ha áttekinthetőbbnek érzi, akkor így is lehet indítani, csak fölösleges tárfoglalás:

```
VezerloOsztaly vezerles = new VezerloOsztaly();
vezerles.Start();
```

A VezerloOsztaly Start () metódusában írjuk meg a vezérlést (azaz a programlogikát). Ez összesen ennyi:

```
public void Start() {
    AdatBevitel();
    Kiiratas("\nRésztvevők:\n");
    Verseny();
    Kiiratas("\nEredmények:\n");
    Eredmenyek();
    Keresesek();
}
```

Az adatbevitelnél kicsit hosszabban elidőzünk. Először megbeszéljük, hogy hogyan lehet billentyűzetről beinni az adatokat, majd – mivel feltehetően roppant unalmas dolog minden újabb futtatás esetén ismét és ismét beolvasni az adatokat, ezért – rátérünk a fájlból való olvasás megtárgyalására. Ha úgy gondolja, hogy a billentyűzetről való olvasást ragyogóan tudja, akkor elég, ha csak átfutja ezt a részt, vagyis csak elolvassa, de nem próbálja ki. A fájlból való olvasást mindenképpen csinálja is meg.

Egyetlen versenyző adatainak beolvasása és az adatok alapján a példány létrehozása igazán gyerekjáték:

```
int sorszam = 1;
string nev, szak;

Console.WriteLine("név: ");
nev = Console.ReadLine();

Console.WriteLine("szak: ");
szak = Console.ReadLine();

Versenyo versenyo = new Versenyo(sorszam, nev, szak);
```

Mivel elégé nyilvánvaló, hogy a magukat megmérni akáró diákok nem a regisztrációkor versenyeznek majd, ezért a létrehozott példányokat tárolni is kell valahol. Erre szolgálnak a tömbök. Létre kell tehát hoznunk egy tömböt, amelyben `Versenyo` típusú példányokat tárolunk.

Logikus feltételezés, hogy előre nem lehet tudni, hányan akarnak regisztrálni, ezért nem lehet `for` ciklussal olvasni. Sajnos még az is elképzelhető, hogy teljes érdektelenségbe fullad a versenyfelhívás, és senki sem jelentkezik, ezért csak elől tesztelő (azaz `while` ciklussal) oldhatjuk meg a beolvasást. Mivel úgy szól a feladat, hogy rajtszámként mindenki a beolvasás sorszámát kapja, ezért szükség lesz egy `sorszam` változóra, melynek értéke növekszik az adatok beolvasása során. (Mivel egy tömb indexelése 0-ról indul, sorszámot viszont 1-ről szokás indítani, ezért ez a sorszám mindenkorábban eggyel nagyobb, mint az index.) Az egyszerűség kedvéért most addig olvassuk az adatokat, míg a név helyére entert nem gépelünk. Ezek után a beolvasás:

```
string nev, szak;
int n = 10;
Versenyo[] versenyzok = new Versenyo[n];
int sorszam = 1;

Console.WriteLine("név: ");
nev = Console.ReadLine();
while (nev != "") {
    Console.WriteLine("szak: ");
    szak = Console.ReadLine();

    versenyzok[sorszam - 1] = new Versenyo(sorszam, nev, szak);
    sorszam++;

    Console.WriteLine("név: ");
    nev = Console.ReadLine();
}
```

Remélhetőleg látja, hogy mi a baj evvel az olvasással. Bár lehet reménykedni benne, hogy senkinek sincs türelme 10 név-szak párost (azaz 20 adatot) begépelni, de ha lenne, akkor ez a kód indextúlcordulás miatt elszállna. Azért választottuk a `while` ciklust, mert nem tudjuk

előre, hogy hány versenyző van, viszont tömböt csak konkrét méretre tudunk deklarálni. Ez bizony ellentmondás.

Ez elég ördögi körnek tűnik, de szerencsére nem az. Van egy nagyon egyszerű megoldás: a Versenyző típusú példányokat nem tömbben, hanem listában tároljuk. Egyelőre elég, ha csak annyit tudunk a listáról, hogy ennek az adatszerkezetnek nem kell előre megadni a méretet. A létrehozásakor üres, és elvileg korlátlan mértékben bővíthető.

A lista elemeire ugyanúgy index alapján hivatkozunk, mint a tömbelemekre. Az eltérés annyi, hogy amint már mondtuk, a létrehozáskor (példányosításkor) üres lista jön létre, amelyet az Add() metódus segítségével bővíthetünk. A lista elemeinek számát nem a Length, hanem a Count tulajdonság adja meg.

Mivel ugyanazt a listát használja az összes metódus, ezért adattagként deklaráljuk. Egy lista deklarálását az esetek túlnyomó többségében célszerű azonnal összekötni a példányosítással is:

```
private List<Versenyző> versenyzök = new List<Versenyző>();
```

Ennek megfelelően javítsuk ki az olvasást:

```
Versenyző versenyző;
string nev, szak;
int sorszám = 1;

Console.WriteLine("név: ");
nev = Console.ReadLine();
while (nev != "") {
    Console.WriteLine("szak: ");
    szak = Console.ReadLine();

    // az adatok alapján létrehozzuk a versenyző példányt
    versenyző = new Versenyző(sorszám, nev, szak);

    // itt adjuk hozzá a listához az aktuális versenyző példányt.
    versenyzök.Add(versenyző);
    // de lehetne így is: versenyzök.Add(new Versenyző(sorszám, nev, szak));
    // ekkor nincs szükség a versenyző változó deklarálására.

    sorszám++;

    Console.WriteLine("név: ");
    nev = Console.ReadLine();
}
```

Figyeljük meg, hogyan olvastunk: beolvassunk egy-egy sort. Ezekben a sorokban felváltva szerepel egy versenyző neve, utána a szakja, majd ismét egy név, egy szak stb. Ezeket az adatokat ugyanilyen módon berakhatjuk egy .txt fájlba is, és beolvásáskor innen vehetjük elő őket. Maga a beolvásás is ugyanilyen, egyetlen dolgot kell csak módosítanunk: nem a konzolról olvasunk, hanem egy fájból, még-

Baranyai Anna építész
Tolnai Edit építész
Somogyi Norbert építész
Fejér Károly gépész

pedig egy olvasócsatornán keresztül. Ez az olvasócsatorna egy `StreamReader` típusú változó, melynek példányosításakor megadhatjuk az olvasandó fájl nevét. (Az eléréssel most ne foglalkozzunk, alapértelmezetten a projekt bin/Debug mappájában keresi, ide rakjuk be. Sőt egyelőre azzal sem foglalkozunk, hogy egyáltalán létezik-e a fájl, helyes adatok vannak-e benne. Természetesen ezek fontos dolgok, de csak később tárgyaljuk alaposabban.) Még egyszerűbb is a beolvasás, hiszen a fájt – a felhasználóval ellentétben – nem kell tájékoztatni arról, hogy épp milyen adatot kellene begépelnünk. Azt, hogy van-e még adat a fájlból, úgy tudjuk ellenőrizni, hogy megnézzük, elérünk-e már az olvasási folyam végére (az olvasócsatorna `EndOfStream` tulajdonsága), és ha még nem, akkor folytatjuk az olvasást.

Bár egyszerűbb esetekben enélkül is működik, az olvasócsatornát az olvasás után le kell zárnai.

Az ennek megfelelően módosított beolvasás, vagyis az `AdatBevitel()` metódus így néz ki:

```
private void AdatBevitel() {  
  
    Versenyo versenyo;  
    string nev, szak;  
    int sorszam = 1;  
  
    StreamReader olvasoCsatorna = new StreamReader("versenyok.txt");  
  
    while (!olvasoCsatorna.EndOfStream) {  
        nev = olvasoCsatorna.ReadLine();  
        szak = olvasoCsatorna.ReadLine();  
  
        // az adatok alapján létrehozzuk a versenyző példányt  
        versenyo = new Versenyo(sorszam, nev, szak);  
  
        // itt adjuk hozzá a listához az aktuális versenyző példányt.  
        versenyok.Add(versenyo);  
        // de lehetne így is: versenyok.Add(new Versenyo(sorszam, nev, szak));  
        // ekkor nincs szükség a versenyo változó deklarálására.  
  
        sorszam++;  
    }  
    olvasoCsatorna.Close();  
}
```

A `StreamReader` osztály nem szerepel az általunk írt névtérben. De sajnos azok között sincs, amelyeket a `using` kulcsszó segítségével beszűrünk a program elején. De semmi baj, pillanatok alatt elérhetővé tehetjük a program elejére írt `using System.IO;` hivatkozás segítségével.

Még egy dolgot kell megemlíteni: hova tegyük az adatfájlt? Mint látható, az előző programkódban útvonal nélkül hivatkoztunk rá. Ezt akkor tehetjük, ha a fájl a program futásakor aktuális mappában van, ez pedig, ahogy már említettük, a projekt bin/Debug mappája.

Reméljük, örül, hogy már tud fájlból olvasni, hiszen ez szemmel láthatóan egyszerűbb, mint billentyűzetről, arról nem is beszélve, hogy ettől kezdve akárhányszor kényelmesen futathatjuk a programot, nem kell ismét és ismét gépelgetnünk az adatokat.

A többi metódusban nincs semmi újdonság. Pontosan ugyanúgy hivatkozunk egy lista elemeire, mint egy tömb elemeire, és pontosan ugyanúgy hivatkozunk egy objektumokat tartalmazó tömb elemeire, mint a primitív típusokból álló tömb esetén.

Kíratakor minden kötelező valamilyen címet írni a kiírt adatok előtt, hiszen a felhasználót tájékoztatni kell arról, hogy mit is lát. A feladat szerint most kétszer is ki kell íratnunk az aktuális adatokat: egyszer a verseny előtt, egyszer utána. Maga a kiíratás egyforma, hiszen minden esetben egy ciklussal végig kell mennünk a versenyzők listáján, és kiíratni a lista összes elemét. Egyedül az adatok elő kiírt címben van eltérés. Ezt célszerű egy metódus-paraméter segítségével megoldani:

```
private void Kiiratas(string cim) {
    Console.WriteLine(cim);
    foreach (Versenyző enekes in versenyzök) {
        Console.WriteLine(enekes);
    }
}
```

A metódus hívása (a Start () metódusban):

```
Kiiratas("\nRésztvevők:\n");
Verseny();
Kiiratas("\nEredmények:\n");
```

A verseny során minden egyes versenyzőt értékel a zsűri, azaz minden egyes zsűritag ad nekik egy-egy véletlenül generált pontot. A zsűritagok számát és az adható pontok határát beolvashatnánk, de most az egyszerűség kedvéért konstansként adjuk meg, viszont úgy, hogy bármikor könnyen lehessen módosítani. Ezért ezt a két változót adattagként deklaráljuk, és mindenkor a deklaráció során megadjuk az értéküket.

```
private int zsuriLetszam = 5;
private int pontHatar = 10;
```

A véletlen értékeket így generálhatjuk:

A Random osztály Next () metódusa a paraméterében lévő értéknél kisebb pozitív egész számokat generál, vagyis ha paraméterként az adható pontok határát adjuk meg, akkor megfelelő nagyságú pontszámot kapunk. Arra azonban figyelnünk kell, hogy a kapott eredmény kisebb lesz a paraméterben szereplő értéknél, vagyis ha a megadott határ „éles”, akkor paraméterként vagy pontHatar + 1-et kell megadnunk, vagy eleve eggyel nagyobb értéket kell adunk a pontHatar változónak.

A Verseny () metódus:

```

private void Verseny() {
    Random rand = new Random();
    int pont;
    foreach(Versenyo versenyzo in versenyzok) {
        // pontoz a zsűri:
        for (int i = 1; i <= zsuriLetszam; i++ ) {
            pont = rand.Next(pontHatar);
            versenyzo.PontotKap(pont);
        }
    }
}

```

Az eredmények meghatározása, kiíratása rutin feladat:

```

private void Eredmenyek() {
    // Nem feltétlenül van szükség erre a metódusra, lehet úgy is,
    // hogy ez a két metódushívás is a Start()-ban szerepel.
    // Esetünkben szubjektív, hogy ki melyik megoldást választja.
    Nyertes();
    Sorrend();
}

private void Nyertes() {
    // kezdőérték beállítása
    int max = versenyzok[0].PontSzam;

    // a legnagyobb érték megállapítása
    foreach (Versenyo enekes in versenyzok) {
        if (enekes.PontSzam > max) {
            max = enekes.PontSzam;
        }
    }

    // a legjobbak kiíratása
    Console.WriteLine("\nA legjobb(ak)\n");
    foreach (Versenyo enekes in versenyzok) {
        if (enekes.PontSzam == max) {
            Console.WriteLine(enekes);
        }
    }
}

private void Sorrend() {
    // rendezés
    Versenyo temp;
    for (int i = 0; i < versenyzok.Count - 1; i++) {
        for (int j = i + 1; j < versenyzok.Count; j++) {
            if (versenyzok[i].PontSzam < versenyzok[j].PontSzam) {
                temp = versenyzok[i];
                versenyzok[i] = versenyzok[j];
                versenyzok[j] = temp;
            }
        }
    }
}

Kiratas("\nEredménytábla\n");
}

```

Megjegyzés: Felmerülhet a kérdés, hogy miért kell „gyalog” rendezni, amikor a listának is, tömbnek is van Sort() metódusa? Azért, mert ezt nem tudjuk közvetlenül alkalmazni objektumtömbök vagy -listák esetén, hiszen azt is meg kellene mondanunk, hogy mi szerint akarunk rendezni. Ha kíváncsi rá, milyen megoldási lehetőségek vannak, akkor nézzen utána pl. itt:

<http://stackoverflow.com/questions/3309188/how-to-sort-a-listt-by-a-property-in-the-object>

A keresésre írt metódust már komment nélkül közöljük:

```

private void Keresesek() {
    Console.WriteLine("\nAdott szakhoz tartozó énekesek keresése\n");
    Console.Write("\nKeres valakit? (i/n) ");
    char valasz;
    while (!char.TryParse(Console.ReadLine(), out valasz)) {
        Console.Write("Egy karaktert írjon. ");
    }
    string szak;
    bool vanIlyen;

    while (valasz == 'i' || valasz == 'I') {
        Console.Write("Szak: ");
        szak = Console.ReadLine();
        vanIlyen = false;

        foreach (Versenyo enekes in versenyzok) {
            if (enekes.Szak == szak) {
                Console.WriteLine(enekes);
                vanIlyen = true;
            }
        }

        if (!vanIlyen) {
            Console.WriteLine("Erről a szakról senki sem indult.");
        }

        Console.Write("\nKeres még valakit? (i/n) ");
        valasz = char.Parse(Console.ReadLine());
    }
}

```

Önálló továbbfejlesztésként találjon ki egyéb keresési szempontokat, és adjon lehetőséget rá, hogy a felhasználó kiválassza, hogy milyen szempont alapján akar keresni.

Szorgalmi: Nézzen utána és próbálja meg szépen formázni a kiíratást (továbbra is a ToString() segítségével):

Eredménytábla				
15	Baranyai Anna	építész	34	pont
10	Zalai Zalán	informatikus	28	pont
5	Nagyvádi Viktor	informatikus	27	pont
5	Békési Emőke	gépész	26	pont
7	Uas Ágnes	környezetmérnök	26	pont
8	Komáromi Konrád	környezetmérnök	25	pont
3	Somogyi Norbert	építész	24	pont
4	Fejér Károly	gépész	23	pont
14	Szabolcs Andrea	informatikus	22	pont
6	Borsodi Gábor	környezetmérnök	21	pont
13	Hevesi Béla	informatikus	21	pont
11	Hajdú Sándor	informatikus	19	pont
2	Talnai Edit	építész	19	pont
9	Pesti Bence	informatikus	17	pont
12	Csongrádi Rózsa	informatikus	16	pont

4. Öröklődés, láthatóság

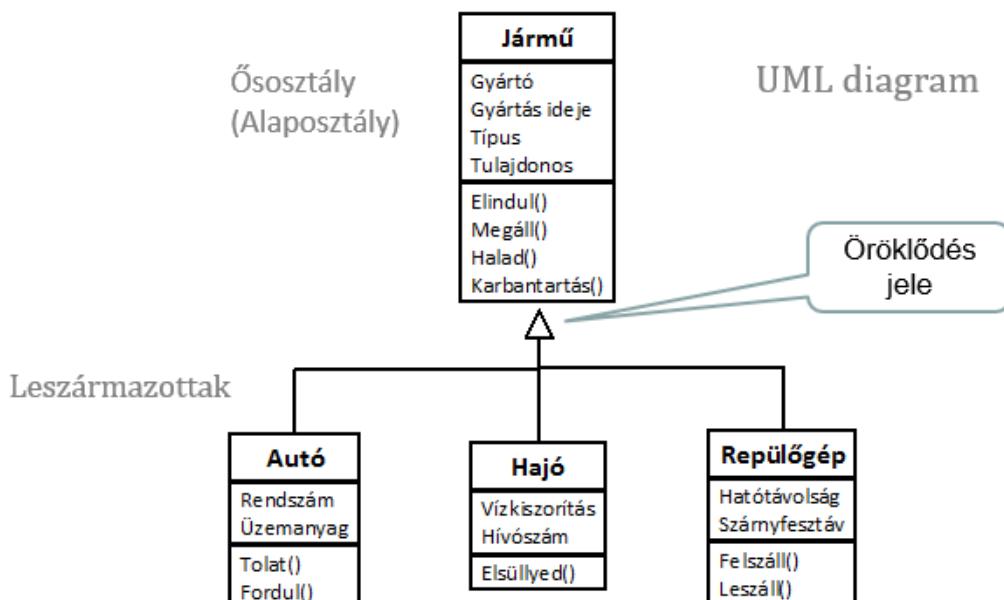
4.1. Elméleti háttér (Szendrői E.)

Specializálással egy már meglévő osztály tulajdonságait és képességeit felhasználva újabb osztályt készíthetünk. A meglévő osztály az ősosztály, a specializált pedig az utódosztály.

A specializálást az osztályok közötti „az egy...” kapcsolatnak, vagy más néven „olyan, mint...” kapcsolatnak is szokás nevezni.

Azt a folyamatot, amikor egy már meglévő osztály felhasználásával (kiterjesztésével) hozunk létre egy osztályt, öröklődésnek nevezzük. Tehát az öröklődés két osztály között értelmezett kapcsolat, öröklődéssel speciális osztályt hozunk létre egy általános osztály tulajdonságaira alapozva. Egy osztály leszármazottai (utódai) öröklik az ősosztály jellemzőit, rendelkeznek ugyanazokkal a mezőkkel, tartalmazzák ugyanazokat a metódusokat. A leszármazottak bővíthetik az ősosztály adatait és metódusait, új mezőket adhatnak a meglévőkhöz, új metódusokat definiálhatnak vagy meglévő metódusok működését módosíthatják.

Egy ősborl több utódosztályt is készíthetünk. Az öröklődési hierarchia mélysége tetszőleges, öröklési láncokat hozhatunk létre. A következő ábra az öröklődés UML diagramban való ábrázolását mutatja. A **Jármű** osztálynak több leszármazottja, utódja van. Az **autó**, a **hajó**, a **repülő** „az egy jármű”, mindegyikük megörökli az ősük, a jármű osztály tulajdonságait, metódusait, és kiegészítik azt a saját speciális tulajdonságaikkal, metódusaikkal. Az utódosztályokban nem tüntetjük fel a megörökült tulajdonságokat, metódusokat, csak a bővíéseket.



13. ábra Öröklődés UML diagram

4.1.1. Az öröklődés megvalósítása C# nyelven

A C# nyelvben az öröklődés operátora a : (kettőspont), melyet az ősosztály neve követ.

```
class Ososztaly
{
    // ősosztály
}

// öröklődés operátora
class Utodosztaly : Ososztaly
{
    // utódosztály
}
```

Az öröklődés során az utódosztály örökli az ősosztály nyilvános (public) és védett (protected) adatmezőit és viselkedését (metódusait, tulajdonságait, indexelőit és eseményeit), melyeket aztán sajátjaként használhat. Privát (private) tagok kizárolag a definiáló osztály számára hozzáférhetők. Az utódosztály bővítheti is a meglévő osztályt, új tagokat definiálhat, illetve felüldefiniálhatja az öröklött, de számára nem megfelelő metódusokat (polimorfizmus).

A C# nyelvben, ha nem adjuk meg explicit módon, az osztály alapértelmezett hozzáférés módosítója az *internal*.

```
namespace TagokHozzafer
{
    class A
    {
        protected int x;
    }
    public class B : A
    {
    }
    class Program
    {
        static void Main(string[] args)
        {
    }
```

Nem fordul be! Miért?

Error List

Description	File	Line	Column	Project
1 Inconsistent accessibility: base class 'TagokHozzafer.A' is less accessible than class 'TagokHozzafer.B'	Program.cs	12	18	TagokHozzafer

Ha egy *internal* hozzáférés módosítóval rendelkező osztályból publikus (*public*) hozzáférés módosítóval ellátott utódosztályt hozunk létre, a fordító hibát jelez. Ezt láthatjuk az előző ábrán. Az *internal* módosító szigorúbb, mint a *public*, tehát nem származtathatunk belőle egy kevésbé szigorú hozzáférés módosítóval rendelkező osztályt.

A C# nyelvben csak egyszeres öröklés van, egy leszármazottnak csak egy közvetlen ōse lehet.

Mivel minden osztály az **Object** osztályból származik, minden osztály megörökli az Object osztály metódusait, melyek a következő táblázatban láthatók.

Metódus	Leírás
Equals()	Meghatározza, hogy két Object példány azonos-e
GetHashCode()	Minden objektum egyedi azonosítóját adja meg.
GetType()	Visszaadja az objektum típusát vagy osztályát
ToString()	Az objektumot reprezentáló stringet ad vissza

A konstruktorkok nem öröklődnek. A leszármazott osztályok először minden a közvetlen ōsosztály konstruktorkat hívják meg, vagyis, ha nem adunk meg másik, akkor az alapértelmezett (default) konstruktort.

Ha az ōsosztályban paraméteres konstruktort definiáltunk, akkor az ōsosztálynak már nincs alapértelmezett konstruktora, ezért a leszármazott osztályokban explicit módon hívni kell a megfelelő konstruktort. Az ōsosztály konstruktornak meghívása a **base** kulcsszóval történik.

Ha van explicit konstruktorthívás (base kulcsszó), akkor a paraméterlistának megfelelő konstruktur hívódik meg az ōsosztályból.

Ha nincs explicit konstruktorthívás, és van az ōsosztályban paraméterek nélküli konstruktur (default konstruktur), akkor azt hívjuk meg.

Ha nincs explicit konstruktorthívás, és az ōsosztályban nincs paraméterek nélküli konstruktur, fordítási hiba keletkezik.

Implicit konverzió

Az ōs és a leszármazottak között „az-egy (is-a)” reláció van. A fordító az öröklődési hierarchia alapján képes az utód példánytípush ōs típusúvá alakítani. Nézzünk egy példát! Deklarálunk egy **Kutya** osztályt, majd egy **Vizsla** leszármazott osztályt, s legyen annak is egy utódosztálya a **MagyarVizsla**. Ez a C# kódban a következőképpen néz ki:

```
class Kutya {  
}  
class Vizsla : Kutya{  
}
```

```
class MagyarVizsla : Vizsla{  
}
```

A következő kód sorban példányosítjuk a MagyarVizsla osztályt, és létrehozunk egy Kutya típusú **eb** referenciát a létrehozott objektumra.

```
Kutya eb=new MagyarVizsla();
```

A fordító implicit módon konvertálta a létrehozott objektumot Kutya típusúvá. Az **eb** objektum valójában **MagyarVizsla** típusú, használhatja annak metódusait, adattagjait. Ez az automatikus konverzió a típuspecializáció egyik fő jellemzője. Arra azonban vigyázunk kell, hogy fordítva ez nem működik, a fordító hibát jelezne. Ősosztály példánya nem konvertálható utód típusúvá.

```
Magyarvizsla vizs = new Kutya();  
vizs.KiVagyTe();  
Console.ReadKey();
```

Már a szövegszerkesztő
hibát jelez!

Error:
Cannot implicitly convert type 'Orokol.Kutya' to 'Orokol.Magyarvizsla'.

Típuskényszerítés

Az előző szakaszban láttuk, hogy a fordító képes implicit típuskonverziót végezni, az utód példánytípushoz átalakítja őstípusúvá. Felmerül a kérdés, hogy fejlesztési időben, tehát a kódíráskor meg tudjuk-e hívni az utód metódusait, hozzáférhetünk-e adattagjaihoz? Nézzük ismét az előző példát! Deklarálunk egy Kutya típusú **vizs** nevű referenciát, ami egy MagyarVizsla objektumpéldányra mutat. Ha meg szeretnénk hívni a Magyarvizsla osztály **Vadasz()** metódusát, nem tudjuk, hiszen csak futáskor, az objektum létrehozásakor fog kiderülni, hogy ez a Kutya típusú **vizs** referencia valójában egy **magyarvizsla** objektumra hivatkozik. Ilyen esetben a kódunkban típuskényszerítést kell alkalmazni.

```
Kutya vizs = new MagyarVizsla();  
//Típuskényszerítés  
vizs.;
```

Equals
GetHashCode
GetType
KiVagyTe
ToString

```
Kutya vizs = new MagyarVizsla();  
//Típuskényszerítés  
((MagyarVizsla)vizs).  
Vadasz();
```

Equals
GetHashCode
GetType
KiVagyTe
ToString
Vadasz

```
Kutya vizs = new MagyarVizsla();  
vizs.KiVagyTe();  
//Típuskényszerítés  
((MagyarVizsla)vizs).Vadasz();
```

Az ábrán jól látszik, hogy csak a típuskényszerítést követően válik elérhetővé az utódosztály **Vadasz()** metódusa.

Megvizsgálhatjuk a típuskényszerítés előtt, hogy milyen típusú az objektum, és a vizsgálat eredményétől függően alkalmazzuk a típuskényszerítést. **IS** operátort használhatjuk erre a célra.

```
Kutya vizes = new MagyarVizsla();
vizes.KiVagyTe();
//Típuskényszerítés
// megvizsgáljuk hogy az objektum Magyarvizsla-e
if (vizes is MagyarVizsla){
    ((MagyarVizsla)vizes).Vadasz();
}
```

Másik megoldás az **as** operátorral:

```
// másik lehetőség a típuskényszerítés helyett
if (vizes is MagyarVizsla){
    // az as operátorral jelezzük, hogy
    // a vizes objektumpéldány viselkedjen
    // MagyarVizsla típusként
    (vizes as MagyarVizsla).Vadasz();
}
```

Az osztályok kódja az alábbiakban tekinthető meg. A kódban definiált metódusokról hamarosan, a következő fejezetben lesz szó.

```
class Kutya {
    public virtual void KiVagyTe(){
        Console.WriteLine("Én egy kutya vagyok");
    }
}
class Vizsla : Kutya {
    public override void KiVagyTe(){
        Console.WriteLine("Én egy vizsla vagyok");
    }
}
class MagyarVizsla : Vizsla {
    public override void KiVagyTe(){
        Console.WriteLine("Én egy magyarvizsla vagyok");
    }
    public void Vadasz() {
        Console.WriteLine("Jó vadász vagyok");
    }
}
```

Egy futási eredmény.

```
Áz a objektum típusa: Orokol.Vizsla
Én egy vizsla vagyok
A b objektum típusa: Orokol.MagyarVizsla
Én egy magyarvizsla vagyok
Én egy kutya vagyok
Én egy magyarvizsla vagyok
Jó vadász vagyok
Jó vadász vagyok
```

A teljes program:

```
static void Main(string[] args)
{
    Kutya a = new Vizsla();
    Console.WriteLine("Az a objektum típusa: " + a.GetType());
    a.KiVagyTe();
    Kutya b = new MagyarVizsla();
    Console.WriteLine("A b objektum típusa: " + b.GetType());
    b.KiVagyTe();
    Kutya eb = new Kutya();
    eb.KiVagyTe();
    Kutya vizs = new MagyarVizsla();
    vizs.KiVagyTe();
    //Típuskényszerítés
    // megvizsgáljuk hogy az objektum Magyarvizsla-e
    if (vizs is MagyarVizsla){
        ((MagyarVizsla)vizs).Vadasz();
    }
    // másik lehetőség a típuskényszerítés helyett
    if (vizs is MagyarVizsla){
        // az as operátorral jelezük, hogy
        // a vizs objektumpéldány viselkedjen
        // MagyarVizsla típusként
        (vizs as MagyarVizsla).Vadasz();
    }
    Console.ReadKey();
}
```

4.1.2. Virtuális metódusok

A felüldefiniálandó metódust virtuális metódusnak nevezzük. Egy metódus felüldefiniálásakor a különböző megvalósításokat ugyanazzal a metódussal látjuk el (ellenértben az elrejtéssel). Egy metódust a **virtual** kulcsszóval jelölhetünk meg virtuálisként. A virtuális metódusokkal ugyanazon metódus különböző verziót hívhatjuk (polimorfizmus).

A leszármazott osztályokban az **override** kulcsszóval mondjuk meg a fordítónak, hogy szándékosan hoztunk létre az ősosztályéval azonos szignatúrájú metódust és a leszármazott osztályon ezt kívánjuk használni mostantól. Egy override–dal jelölt metódus automatikusan virtuális is lesz, így az Ő leszármazottai is átdefiniálhatják a működését.

A virtuális metódusok használatakor az alábbi szabályokat kell betartanunk:

- Nem deklarálhatunk privát metódust a *virtual* és az *override* kulcsszavakkal.
- A két metódus szignatúrájának azonosnak kell lennie.
- A két metódusnak azonos hozzáféréssel kell rendelkeznie.
- Csak virtuális metódusokat bírálhatunk felül.
- Ha az utódosztály nem deklarálja a metódust az *override* kulcsszóval, akkor nem definiálja felül az ősosztálybeli metódust (elrejtést eredményez).

- Az *override* metódusok hallgatólagosan virtuálisak, és felüldefiniálhatók egy másik utódosztályban.

A korábban már látott példa a virtuális metódusok használatát mutatja be.

A virtuális metódusokkal ugyanazon metódus különböző verziót hívhatjuk, a futtatórendszerben dinamikusan meghatározott objektumtipusnak megfelelően.

```
class Kutya {
    public virtual void KiVagyTe() { Console.WriteLine("En egy kutya vagyok"); }
}
class Vizsla : Kutya {
    public override void KiVagyTe() { Console.WriteLine("En egy vizsla vagyok"); }
}
class MagyarVizsla : Vizsla {
    public override void KiVagyTe() { Console.WriteLine("En egy magyar vizsla vagyok"); }
}

Kutya eb = new MagyarVizsla(); // megengedett
eb.KiVagyTe(); // "En egy magyar vizsla vagyok"

Kutya eb2 = new Vizsla(); // megengedett
eb2.KiVagyTe(); // "En egy vizsla vagyok"
```

4.1.3. Láthatóság, védelem

Az utód csak azt láthatja, amit az ős megenged neki. A nyilvános (**public**) tagok, metódusok bárhonnét, külső osztályokból is láthatók. A védett (**protected**) módosítóval rendelkező tagok, metódusok az őket definiáló osztályban és az utódosztályból is elérhetők. A privát (**private**) adattagok és metódusok csak az őket definiáló osztályban láthatók, másik osztályban, így az utódosztályban, nem.

4.1.4. Absztrakt osztályok

Egy absztrakt osztályt nem lehet példányosítani. A létrehozásának célja az, hogy közös felületet biztosítsunk a leszármazottainak. Absztrakt osztályt a következőképpen definiálhatunk:

```
abstract class Allat{
    abstract public Koir();
}

class Kutya : Allat{
    public override Koir(){
        Consol.WriteLine("Ez a kutya osztály");
    }
}
```

A fenti kódrészletből látható, hogy mind az osztályt, mind a metódust absztraktként deklaráltuk, ugyanakkor a metódus (látszólag) nem virtuális és nincs definíciója. Valójában az absztrakt metódus egyben virtuális is, ezért a leszármazottban felül kell definiálni.

Említsünk meg néhány fontos szabályt az absztrakt osztályokkal kapcsolatosan:

- absztrakt osztályt nem lehet példányosítani,
- kötelező származtatni belőle,
- absztrakt metódusnak nem lehet definíciója, csak deklaráljuk,
- az utódnak definiálnia kell az öröklött absztrakt metódusokat. Az `override` kulcsszó segítségével tudjuk definiálni (hiszen virtuálisak, még ha nem is látszik).

Absztrakt osztály tartalmazhat nem absztrakt metódusokat is. Amennyiben egy osztálynak van legalább egy absztrakt metódusa, az osztályt is absztraktként kell jelölni.

4.1.5. Lezárt osztályok és lezárt metódusok

Egy osztályt lezárhatsunk (`sealed`), azaz megtilthatjuk, hogy új osztályt származtassunk belőle:

```
sealed Class Kutya{  
}  
class Pasztor : Kutya{      // ez nem lehetséges,  
                           // a Kutya osztály sealed osztály  
}
```

A **String** osztály **sealed**, így nem tudunk a String osztályból utódosztályt létrehozni.

Egy **metódust** is deklarálhatunk lezártként, ekkor a leszármazottak már nem definiálhatják át a működését:

Tekintsük át, milyen kapcsolat van a `virtual`, `override` és `sealed` módosítók között, a metódusokat tekintve:

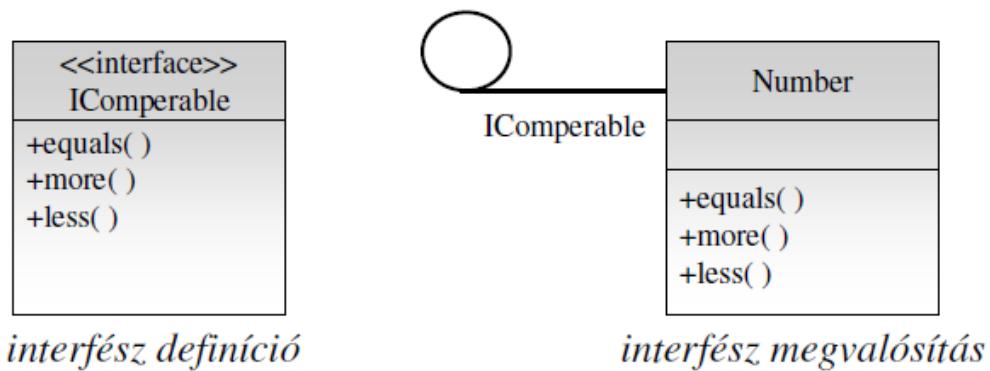
- Egy **virtuális** metódus a metódus *első megvalósítása*.
- Egy **felüldefiniáló (override)** metódus a metódus *egy másik megvalósítása*.
- Egy **lezárt (sealed)** metódus a metódus *utolsó megvalósítása*.

4.1.6. Interfészek

A C# nyelvben minden osztálynak csak egyetlen közvetlen őse lehet, ugyanakkor sok esetben hasznos lenne, ha egy osztály több helyről is örökölhetne tulajdonságokat, metódusokat.

Egy osztálynak csak egy közvetlen őse lehet, de több interfészt is megvalósíthat. Az *interfész* publikus absztrakt metódus deklarációk, tulajdonságok, indexelők és események összessége.

UML jelölése:



Az interfész metódusai üres törzzsel vannak deklarálva, tagjai nem tartalmazhatnak hozzáférés módosítót. Önmagukban nem tudjuk használni, implementálni kell őket. Az implementációt egy-egy osztály végzi. Egy osztály több interfészt is implementálhat. Az interfész neve konvenció szerint nagy i betűvel kezdődik.

Szintaxisa:

[módosító] interface *Interfészneve* {

// absztrakt metódusok;

}

Amennyiben egy osztályból és interfésemből is származtatunk, akkor a felsorolásnál az ősosztály nevét kell előrevenni, utána jönnek az interfések:

class Utod : OsOsztaly, Interfész1, Interfész2,...

{

// az osztály tagjai

}

Általában akkor használunk interfészt, amikor logikailag nem összetartozó osztályok egyforma metódust használnak. A következő feladatban, az *IAGE* interfészt valósítja meg a *Szemely* és a *Fa* osztály, s kiszámolja a személy, illetve a fa életkorát.

Az interfész:

```
namespace InterfeszKorok
{
    public interface IAge
    {
        int Age { get; }
        string Name { get; }
    }
}
```

A személy osztály kódja, amelyben megvalósítja az IAGE interfészt és kiszámolja a kor, Age tulajdonság értékét, valamint a személy nevét is megadja.

```
public class Szemely : IAge
{
    private string vezNev;
    private string kerNev;
    private int szulEv;

    // konstruktor
    public Szemely( string vezNev, string kerNev, int szulEv )
    {
        this.vezNev = vezNev;
        this.kerNev=kerNev;

        if ( szulEv > 0 && szulEv <= DateTime.Now.Year )
            this.szulEv = szulEv;
        else
            this.szulEv = DateTime.Now.Year;
    }
    public int Age {
        get { return DateTime.Now.Year - szulEv; }
    }
    public string Name {
        get { return vezNev + " " + kerNev; }
    }
}
```

A Fa osztály szintén megvalósítja az IAge interfészt.

```
public class Fa : IAge
{
    private int gyuruk;
    public Fa(int ultetesEve) {
        this.gyuruk = DateTime.Now.Year - ultetesEve;
    }
    public void GyuruNoveles() {
        gyuruk++;
    }
    public int Age {
        get { return gyuruk; }
    }
    public string Name {
        get { return "Fa"; }
    }
}
```

Az interfészek használatakor néhány fontos alapelvet be kell tartanunk, melyek a következők:

- Az interfésznek nem lehet konstruktora, sem destruktora.
- Az interfésznek nem lehetnek mezői.
- Hozzáférés módosítókat nem használhatunk, hallgatólagosan minden metódusa publikus.

A .NET Framework osztálykönyvtára néhány fontos interfészt tartalmaz, a legfontosabb az IComparable, az IComponent, az IDisposable és az IEnumerator interfész.

Azok az osztályok, amelyek implementálják az IComparable interfészt, megvalósítják a CompareTo() metódust, amely összehasonlítja a metódust hívó objektumot a paraméterként kapott objektummal.

Az IComponent interfészt minden olyan osztály megvalósítja, amely komponenst reprezentál.

Az IDisposable interfészt azok az osztályok valósítják meg, amelyeknek erőforrások felszabadítását kell biztosítaniuk.

Az IEnumerator interfész megvalósításával egy gyűjtemény elemein haladhatunk végig.

4.2. Gyakorlati példák (Achs Á.)

4.2.7. Állatverseny folytatása

Vészesen közeleg az állatverseny időpontja, és a megrendelő most jött rá, hogy módosítani kellene a programunkat (a megrendelők már csak ilyenek ☺).



Hiába dugjuk az orra alá az eredeti kérését, mégpedig ezt:

„Az állatmenhely-alapítvány kisállatversenyt rendez. Mindegyik **állat** regisztrálásakor meg kell adni az állat *nevét* és a *születési évét*. Ezek a verseny során nyilván nem változhatnak. Mindegyikötüket pontozzák, pontot kapnak a *szépségükre* és a *viselkedésükre* is.

A *pontszám* meghatározásakor figyelembe veszik a korukat is (csak év): egy egységesen érvényes *maximális kor* fölött 0 pontot kapnak, alatta pedig az életkor arányában veszik figyelembe a szépségre és a viselkedésre adott pontokat. Minél fiatalabb, annál inkább a szépsége számít, és minél idősebb, annál inkább a viselkedése. (Ha pl. 10 év a maximális kor, akkor egy 2 éves állat pontszáma: (10 – 2) a szépségére adott pontok + 2 a viselkedésre kapott pontok.)”

Ő közli, hogy bocsánat, tévedett, és kéri a módosítást. Kiderült ugyanis, hogy a versenyen kutyák és macskák vesznek részt, nem teljesen egyforma feltételekkel. A regisztrációra és a pontozásra való előírás nagyjából marad, de ezeket a módosításokat kéri:

Kutyák esetén a gazdához való *viszonyt* is pontozzák. Ez hozzáadódik a szépségért és viselkedésért kapott pontokhoz, de ezt a viszonyPontot még a verseny előtt adja a zsűri, és csak akkor vehet részt a kutyá a versenyen, ha már túlesett ezen az előfeltételen. Ha nincs viszonyPontja, akkor a végső pontszáma nulla lesz.

Mivel kutyák és macskák együtt szerepelnek, ezért csak olyan macskák versenyezhetnek, akiknek *van* macskaszállító dobozuk. A doboz létét már a regisztráció során be kell jelenteni, de ez a verseny pillanatáig módosítható (vagyis esetleg utólag is pótolhatják ezt a dobozt, vagy persze, el is veszíthetik). Ha a verseny pillanatában nincs ilyen doboz, akkor az ő végső pontszáma is nulla lesz.

Bár a megrendelő csak ennyit mondott, de azért figyelmeztessük rá, hogy így nem lehet egyértelműen azonosítani az állatokat, hiszen miért ne lehetne köztük két azonos nevű. Most állapodunk meg abban, hogy mindenki kap egy rajtszámot, mégpedig a regisztrálás sorrendjének megfelelő értéket.

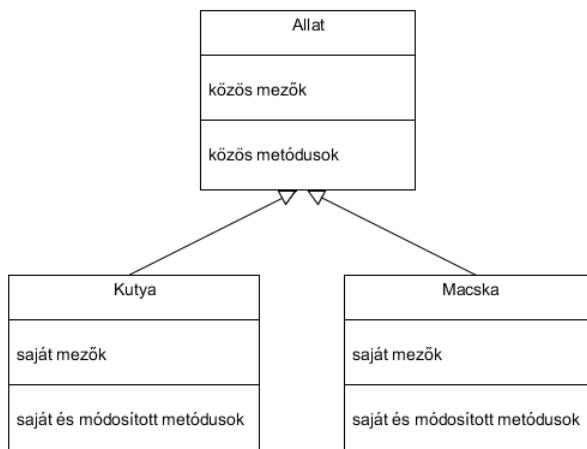
Minden állat *ToString()* metódusa így nézzen ki: rajtszám, kutya/macska neve, pontszáma.

1. Először próbáljuk ki az elkészült osztályokat két konkrét példányra.
2. Utána regisztrálunk valahány állatot (vegyesen kutyákat és macskákat), majd versenyezzük őket. A regisztráció után is és a verseny után is írassuk ki az adataikat. Az adatokat fájlból olvassuk. Azt is gondolja végig, milyen szerkezetben kellene megadni az adatfájlt.

Megoldásjavaslat

Természetesen minél kevesebb pluszmunkával szeretnénk módosítani a már meglévő megoldást, ezért nem kezdjük előlről az egészet, hanem végiggondoljuk, mit lehetne felhasználni a már meglévőből. No és persze, a kódismétlés is kerülendő. Ezért célszerű igénybe venni az öröklődés fogalmát. Tudjuk ugyanis, hogy a kutya is, macska is állat, és vannak olyan feltételek, amelyek mindenktőre vonatkoznak. Ezeket célszerű csak egyszer megfogalmazni. A közös tulajdonságok kerülnek majd az ősosztályba (`Allat`), az utódosztályokban (`Kutya`, `Macska`) pedig csak a módosításokat kell majd leírnunk.

Az osztályszerkezet sematikus UML ábrája:



Induljunk ki tehát a már meglévő `Allat` osztályból. Erre kétféle megoldást is megbeszélünk. Ismételjük át minden UML ábrát:

Allat
- string nev
- int szuletesiEv
- int szepsegPont
- int viselkedesPont
- int pontSzam
- int aktualisEv
- int korHatar
+ Allat(string nev, int szuletesiEv)
+ int Kor()
+ void Pontozzak(int szepsegPont, int viselkedesPont)
+ string ToString()
tulajdonságok

Allat
- string nev
- int szuletesiEv
- int szepsegPont
- int viselkedesPont
- int aktualisEv
- int korHatar
+ Allat(string nev, int szuletesiEv)
+ int Kor()
+ void Pontozzak(int szepsegPont, int viselkedesPont)
+ int PontSzam()
+ string ToString()
tulajdonságok

Mint látjuk, az a különbség, hogy a bal oldaliban a pontszámot adattagként kezeltük és a Pontozzak() metódusban számoltuk ki az értékét, a jobb oldaliban pedig egy külön metódust írtunk a kiszámítására. Mindkét megoldás jó, mindenki megoldást alapul tudjuk venni, csak eltérő lesz a továbblépés (mint ahogy a kiindulás is az volt ☺).

Minden változatot megbeszéljük. Kezdjük a bal oldalival.

A kutyanak is, macskának is van neve, születési éve, mindenkitől kap szépségpontot és viselkedéspontot, sőt még is engedjük, hogy ezeket bármikor le is lehessen kérdezni (ezért definiáljuk mezőként és nem csak egyszerű paraméterként), mindenkitől kiszámolhatjuk a korát, és mindenkitől ugyanaz a korhatár-előírás vonatkozik. Vagyis az Allat osztály nev, szuletesiEv, szepsegPont, viselkedesPont, aktualisEv, korHatar mezőit nem kell (nem szabad) újraírnunk az utód osztályokban, és az életkort is ugyanúgy számoljuk, vagyis a Kor() metódust sem írjuk újra.

Mi a helyzet a Pontozzak() metódussal? Itt már van egy kis probléma, mert nem egyformán pontozzák őket. Ugyanakkor azonban a szépségpont és viselkedéspont alapján kiszámított rész most is egyforma. Ezt úgy lehet megoldani, hogy a pontozásnak ezt a részt a közös ősben írjuk meg, az utód osztályokban pedig hozzárakjuk az eltérő részleteket, vagyis az utód osztályban módosítjuk (felüldefiniáljuk) ezt a metódust. A Macska osztályban nincs is gond, hiszen csak annyi a módosítás, hogy egy feltételtől tesszük függővé, hogy meghívjuk-e az ősosztály Pontozzak() metódusát, de a Kutya osztályban a kiszámítás módja is változik, az eddigi értékhez hozzáadódik még egy pontszám. Igen ám, de minden adattag private, és nem is engedtük meg, hogy a pontSzam változó értékét a Pontozzak() metóduson kívül bárki más megváltoztathassa. Most kénytelenek vagyunk engedni ebből a szigorú megszorításból, és megengedni azt, hogy az utód mégiscsak módosíthassa a pontSzam értékét. Emiatt a változóhoz protected set tulajdonságot kell majd rendelnünk.

Lássuk, mi új lesz a Kutya osztályban!

Lesz két új mező, az egyik a gazdaViszonyPont (ehhez nem feltétlenül kellene mezőt rendelnünk, csak akkor, ha a teljes pontszámtól függetlenül is meg akarjuk kérdezni az értékét), illetve egy logikai változó, amelyik azt mutatja, hogy kapott-e már ilyen pontot. (Ez lesz a kapottViszonyPontot változó.) Szükség lesz még a ViszonyPontozas() metódusra, ez kéri be a gazdaViszonyPont értékét, és módosítanunk kell, azaz felül kell definiálnunk a Pontozzak() metódust.

Gondoljuk végig a Macska osztályt is!

Itt kell egy logikai változó, melynek értéke mutatja, hogy van-e macskaszállító doboz (ezt vanMacskaSzallitoDoboz néven nevezzük). Itt is felül kell definiálnunk a Pontozzak() metódust.

Fontos tudnunk, hogy az utód osztály konstruktörét mindenkor újra kell gondolnunk. A Kutya osztály konstruktörának ugyanazok a paraméterei, mint amelyeket az Allat osztályban is megadtunk, hiszen egy kutya regisztrálásakor csak ezeket az adatokat kérünk. A Macska osztály konstruktörét azonban ki kell bővítenünk a vanMacskaSzallitoDoboz paraméter-

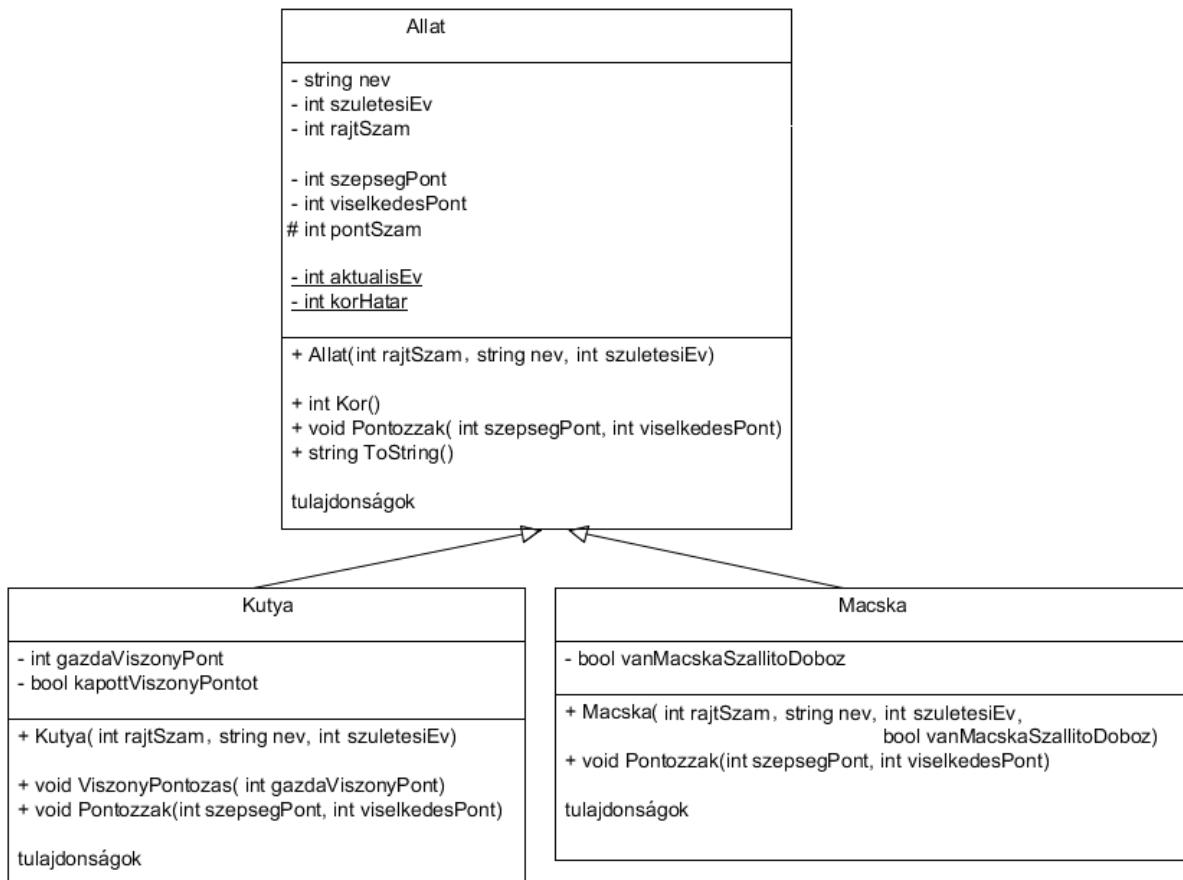
rel, hiszen esetükben már a regisztráció során nyilatkozni kell arról, hogy van-e ilyen doboz vagy sincs.

Kérdés még a `ToString()` sorsa. A megrendelő azt is kéri, hogy az állat neve mellé írjuk ki, kutyáról vagy macskáról van-e szó. Első hallásra úgy tűnik, hogy emiatt a `ToString()` metódust is át kell alakítanunk, és felül kell definiálnunk az utód osztályokban. De ha ügyes módon választunk osztályneveket, akkor egyszerűbben is megoldható a feladat. Elég, ha csak az ősosztály `ToString()` metódusát alakítjuk át, mégpedig úgy, hogy a visszaadott stringbe foglalja bele az aktuális osztály nevét is. Így Kutya példány esetén a kutya szót írja ki, Macska példány esetén a macska szót (feltéve, hogy arra is odafigyelünk, hogy alakítsa a nevet csupa kisbetűssé).

Végül ne feledkezzünk el a saját javaslatunkról, vagyis, legyen minden állatnak rajtszáma. Mivel ez minden állatra ugyanúgy vonatkozik, ezért ezt most úgy oldjuk meg, hogy az `Allat` osztályban felveszünk egy újabb mezőt `rajtSzam` néven. Ez a mező a konstruktőrben kap értéket, amely a versenykiírásnak megfelelően később nem változhat.

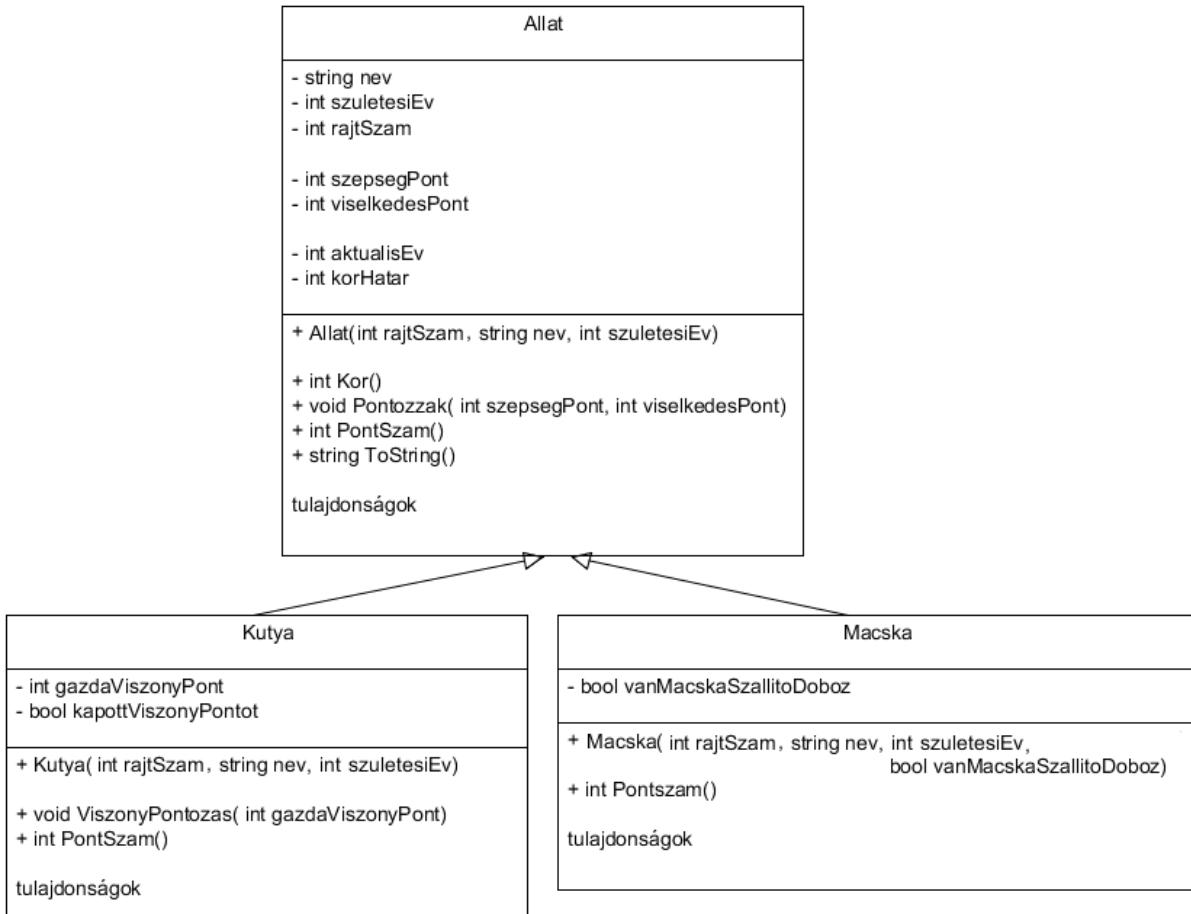
(Megjegyzés: ez a fajta megoldás nem garantálja azt, hogy a rajtszámok egyediek legyenek, de egyelőre megelégszünk ilyen megoldással.)

Ezek után lássuk az UML ábrát:



Láttuk, hogy egy kicsit körülményessé tette a megoldást az, hogy a pontszámot mezőként definiáltuk. Várhatóan egyszerűsíti a kódot, ha az ősosztályban külön metódust írunk a pontszám kiszámítására (ez lesz a `PontSzam()` metódus). Ekkor az ős `Pontozzak()` metódusa változatlan marad (csak arra szolgál, hogy a zsúri megadhassa a pontszámokat), és a `PontSzam()` metódust kell majd felüldefiniálnunk minden utódban.

Az ennek megfelelően módosított UML:



Ez utóbbi megoldást beszéljük meg részletesen, de a fejezet végén kommentezve láthatja majd az első elképzélés kódolt változatát is.

Az `Allat` osztály megírása már nem jelenthet problémát. Ha mégis, akkor alaposan végig-gondolva oldja meg újra az erre vonatkozó korábbi feladatokat.

Ennek ellenére ide kerül majd a kód, de előbb beszéljünk meg egy kis újítást. Mivel az egységezés elve alapján a mezőket `private` láthatóságuként használjuk, ezért, ahogy tudja is, tulajdonságokat definíálunk hozzájuk, és ezeken keresztül állítjuk be, hogy melyeket engedünk – és esetleg milyen feltételek mellett – lekérdezhetővé vagy módosíthatóvá tenni. Ha csak az elérhetőséget akarjuk szabályozni, de ezt nem kötjük semmiféle különleges feltételhez, akkor a tulajdonságokat egyszerűbben is definiálhatjuk az úgynevezett automatikus tulajdonságok segítségével. Ennek során lehetőség van rá, hogy összevonjuk a

meződeklarálást a tulajdonságok definiálásával. Ilyenkor csak a tulajdonságot definiáljuk, és csak jelezzük, hogy milyen módon lehet hozzáférni a mező értékéhez. A

```
public string Nev { get; private set; }
```

automatikus tulajdonság-beállítás például azt jelenti, hogy definiáltuk a `Nev` tulajdonságot, melynek értékét bármikor lekérdezhetjük, de nem módosíthatjuk azt.

Mivel az automatikus tulajdonságdeklaráció helyettesíti a mező deklarálását is, ezért most nem lesz `nev` nevű mező, nem is hivatkozhatunk rá az osztályon belül sem. Semmi baj, ezen túl nem a mezőre, hanem a tulajdonságra hivatkozunk az osztályon belül is.

Még egy dolgot meg kell tárgyalni: lehetővé kell tennünk, hogy az utódokban átírhassuk a `PontSzam()` metódust. Ehhez a metódust virtuálissá kell tenni, azaz el kell látni a `virtual` módosítóval.

Ezek alapján az `Allat` osztály:

```
class Allat {

    // mezők és tulajdonságok helyett automatikus tulajdonságok:
    public string Nev { get; private set; }
    public int SzuletesiEv { get; private set; }
    public int RajtSzam { get; private set; }

    public int SzepsegPont { get; private set; }
    public int ViselkedesPont { get; private set; }

    public static int AktualisEv { get; set; }
    public static int KorHatar { get; set; }

    // konstruktor
    // mivel nincsenek mezők, értékkedáskor a tulajdonság kap értéket
    public Allat(int rajtSzam, string nev, int szuletesiEv) {
        this.RajtSzam = rajtSzam;
        this.Nev = nev;
        this.SzuletesiEv = szuletesiEv;
    }

    // metódusok
    public int Kor() {
        return AktualisEv - SzuletesiEv;
    }

    // örökölhetővé kell tenni, ezért átírjuk virtual-ra
    // és itt is a tulajdonságokra hivatkozunk
    public virtual int PontSzam() {
        if (Kor() < KorHatar) {
            return ViselkedesPont * Kor() + SzepsegPont * (KorHatar - Kor());
        }
        return 0;
    }
}
```

```

    public void Pontozzak(int szepsegPont, int viselkedesPont) {
        this.SzepsegPont = szepsegPont;
        this.ViselkedesPont = viselkedesPont;
    }

    // A ToString() így majd kisbetűsen kiírja az osztályneveket is.
    public override string ToString() {
        return RajtSzam + ". " + Nev + " nevű "
            + this.GetType().Name.ToLower()
            + " pontszáma: " + PontSzam() + " pont";
    }
}

```

Lássuk a `Kutya` osztályt! Ennek kapcsán két dolgot kell megbeszélnünk: a konstruktur és a felüldefiniált metódus kérdését.

Azt hogy az osztály az állat osztály leszármazottja, ilyen módon jelöljük: `class Kutya : Allat`

Az űsre a `base` kulcsszó segítségével hivatkozunk, így a konstruktur:

```

public Kutya(int rajtSzam, string nev, int szulEv) :
    base(rajtSzam, nev, szulEv) {
}

```

Esetünkben ez annyit jelent, hogy a konstruktur hivatkozik az űs konstrukturára, és teljes mértékben elfogadja az általa létrehozott példányt.

Azt pedig, hogy egy metódus felülírja az űs metódusát, a `ToString()` használatakor már megismert `override` módosítóval jelezzük.

Ezek után az osztály kódja:

```

class Kutya : Allat {

    public int GazdaViszonyPont { get; private set; }
    public bool KapottViszonyPontot { get; private set; }

    public Kutya(int rajtSzam, string nev, int szulEv) :
        base(rajtSzam, nev, szulEv) {
    }

    public void ViszonyPontozas(int gazdaViszonyPont) {
        this.GazdaViszonyPont = gazdaViszonyPont;
        KapottViszonyPontot = true;
    }

    public override int PontSzam() {
        int pont = 0;
        if (KapottViszonyPontot) {
            pont = base.PontSzam() + GazdaViszonyPont;
        }
        return pont;
    }
}

```

A Macska osztály:

```
class Macska : Allat {  
  
    public bool VanMacskaSzallitoDoboz { get; set; }  
  
    public Macska(int rajtSzam, string nev, int szulEv, bool vanMacskaSzallitoDoboz) :  
        base(rajtSzam, nev, szulEv) {  
            this.VanMacskaSzallitoDoboz = vanMacskaSzallitoDoboz;  
        }  
  
    public override int PontSzam() {  
        if (VanMacskaSzallitoDoboz) {  
            return base.PontSzam();  
        }  
        return 0;  
    }  
}
```

Az osztály konstruktora hivatkozik az ōs konstruktóra, segítségével létrehoz az objektumból annyit, amennyit tud, majd kiegészít még a `vanMacskaSzallitoDoboz` változó inicializálásával.

A `Main()` metódus feladata most is csak annyi, hogy elindítsa a vezérlést:

```
class Program {  
  
    static void Main(string[] args) {  
        new Vezerles().Start();  
  
        Console.ReadKey();  
    }  
}
```

Azt szeretnénk majd elérni, hogy a jelentkezés sorrendjében (azaz vegyesen kutyákat, macskákat) regisztrálhassuk az állatokat, majd a verseny során mindenkit pontozzanak. Ezért deklarálnunk kellene az `Allat` típusú változókból álló listát, először azonban csak annyit oldunk meg, hogy létrehozunk egy-egy példányt és versenyeztetjük őket. Ezt teszi majd a `Vezerles` osztály `Proba()` metódusa.

A `Vezerles` osztály „eleje”:

```

class Vezerles {

    // ez nem kell a próbához
    private List<Allat> allatok = new List<Allat>();

    public void Start() {

        Allat.AktualisEv = 2015;
        Allat.KorHatar = 10;

        // először csak ennyi
        Proba();

        // önállóan oldja meg a többöt
        Regisztracio();
        Kiiratas("A regisztrált versenyzők");
        Verseny();
        Kiiratas("A verseny eredménye");
    }
}

```

A Proba metódusban először két konkrét állatot (egy kutyát és egy macskát) versenyeztetünk:

```

private void Proba() {
    Allat allat1, allat2;

    string nev1 = "Pamacs", nev2 = "Bolhazsák";
    int szulEv1 = 2010, szulEv2 = 2011;
    bool vanDoboz = true;
    int rajtSzam = 1;

    int szepsegPont = 5,
        viselkedesPont = 4,
        viszonyPont = 6;

    allat1 = new Kutya(rajtSzam, nev1, szulEv1);
    rajtSzam++;

    allat2 = new Macska(rajtSzam, nev2, szulEv2, vanDoboz);

    Console.WriteLine("A regisztrált állatok: ");
    Console.WriteLine(allat1);
    Console.WriteLine(allat2);

    // verseny:
    allat1.Pontozzak(szepsegPont, viselkedesPont);
    allat2.Pontozzak(szepsegPont, viselkedesPont);

    Console.WriteLine("\nA verseny eredménye: ");
    Console.WriteLine(allat1);
    Console.WriteLine(allat2);
}

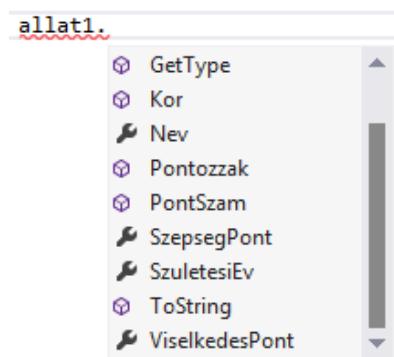
```

Ha eddig kipróbáltuk, akkor tapasztalhatjuk, hogy szegény Pamacs csak 0 pontot kapott, mert elfelejtették zsűriztetni a gazdájához való viszonyát. (Egyébként a Visual Studio is jelzi, hogy nem használtuk a `viszonyPont` változót, ezért van aláhúzva.)

Nosza, pótoljuk gyorsan:

Kiderül, hogy nem tudjuk, mert nem érjük el a `ViszonyPontozas()` metódust. Jogosan, hiszen az a `Kutya` osztályhoz tartozik, mi viszont `Allat` típusúra deklaráltuk a példányt. Megoldás lehetne az, ha `Kutya` típusra deklarálnánk, de akkor nem tudnánk vegyesen regisztrálni őket, vagyis nem tudnánk egyetlen listában tárolni a példányokat.

Ezért más megoldáshoz folyamodunk: típuskényszerítést alkalmazunk: Előbb rákényszerítjük az `allat1` nevű változóra, hogy `Kutya` típusként viselkedjen, és erre a példányra hívjuk meg a `ViszonyPontozas()` metódust:



```
((Kutya)allat1).ViszonyPontozas(viszonyPont);
```

(A `(Kutya)allat1` jelenti azt, hogy létrehoztuk a példányt, a körülötte lévő zárójel pedig azt, hogy erre a példányra alkalmazzuk a metódust.)

Van azonban evvel egy kis bibi. Mégpedig az, hogy mi van, ha ezt írjuk?

```
((Kutya)allat2).ViszonyPontozas(viszonyPont);
```

Látszólag semmi. Nem kapunk szintaktikus hibaüzenetet – miért is kapnánk –, de futáskor elszáll a program, hiszen az `allat2` nem kutya. A futáskori hiba a lehető leggonoszabb dolog, ezt amennyire csak lehet, kerülni kell. Meg is van rá a módunk: típuskényszerítés előtt ellenőrizni kell, hogy a példány valóban `Kutya` típusú-e. Ezt az `is` operátorral tehetjük meg:

```
// helyesen:
if (allat1 is Kutya) {
    ((Kutya)allat1).ViszonyPontozas(viszonyPont);
}

// vagy így is lehet:
if (allat1 is Kutya) {
    (allat1 as Kutya).ViszonyPontozas(viszonyPont);
}
```

Mint látható, a típuskényszerítésnek van egy, talán egyszerűbb változata is: az `as` operátor segítségével közöljük, hogy az `allat1` példány viselkedjen `Kutya` típusként.

A `Proba()` metódus verseny részének javított változata:

```

// verseny:
allat2.Pontozzak(szepsegPont, viselkedesPont);

if (allat1 is Kutya) {
    (allat1 as Kutya).ViszonyPontozas(viszonyPont);
}

allat1.Pontozzak(szepsegPont, viselkedesPont);

```

Ezek után feltételezhetően önállóan is meg tudja írni a hiányzó metódusokat, de egy kis segítség:

A beolvasáshoz használt adatfájl javasolt szerkezete: soronként egy-egy adat: a kutya vagy macska szó; az állat neve, születési éve, és ha macska, akkor true vagy false jelzi azt, hogy van-e macskaszállító doboza.

```

private void Regisztracio() {
    StreamReader olvasoCsatorna = new StreamReader("allatok.txt");

    string fajta, nev;
    int rajtSzam = 1, szulEv;
    bool vanDoboz;

    // Addig olvasunk, amíg el nem értük a fájl végét.
    while (!olvasoCsatorna.EndOfStream) {
        fajta = olvasoCsatorna.ReadLine();
        nev = olvasoCsatorna.ReadLine();
        szulEv = int.Parse(olvasoCsatorna.ReadLine());

        // létrehozzuk a példányokat és hozzáadjuk az allatok listához.
        if (fajta == "kutya") {
            allatok.Add(new Kutya(rajtSzam, nev, szulEv));
        }
        else {
            vanDoboz = bool.Parse(olvasoCsatorna.ReadLine());
            allatok.Add(new Macska(rajtSzam, nev, szulEv, vanDoboz));
        }
        rajtSzam++;
    }
    olvasoCsatorna.Close();
}

```

kutya	
Bodri	
2010	
kutya	
Rexi	
2013	
macska	
Cirmos	
2011	
true	
kutya	
Pamacs	
2012	
macska	
Bolhazsák	
2013	
true	
kutya	
Buksi	
2010	
macska	
Picúr	
2012	
false	

A verseny:

```

private void Verseny() {
    Random rand = new Random();
    int hatar = 11;
    foreach (Allat item in allatok) {
        if (item is Kutya) {
            (item as Kutya).ViszonyPontozas(rand.Next(hatar));
        }
        item.Pontozzak(rand.Next(hatar), rand.Next(hatar));
    }
}

```

Végül a fejezet elején tett ígéretnek megfelelően, a másik változat kódja

```
class Allat {  
  
    // mezők és tulajdonságok helyett automatikus tulajdonságok:  
  
    public string Nev { get; private set; }  
    public int SzuletesiEv { get; private set; }  
    public int RajtSzam { get; private set; }  
    public int SzepsegPont { get; private set; }  
    public int ViselkedesPont { get; private set; }  
  
    // az egyik utóban újra akarjuk számolni, ezért protected  
    public int PontSzam { get; protected set; }  
  
    public static int AktualisEv { get; set; }  
    public static int KorHatar { get; set; }  
  
    // konstruktur - nincs mező, ezért a tulajdonságok kapnak értéket  
    public Allat(int rajtSzam, string nev, int szuletesiEv) {  
        this.RajtSzam = rajtSzam;  
        this.Nev = nev;  
        this.SzuletesiEv = szuletesiEv;  
    }  
  
    // metódusok  
    public int Kor() {  
        return AktualisEv - SzuletesiEv;  
    }  
  
    // örökölhetővé kell tenni, ezért átírjuk virtual-ra  
    public virtual void Pontozzak(int szepsegPont, int viselkedesPont) {  
        this.SzepsegPont = szepsegPont;  
        this.ViselkedesPont = viselkedesPont;  
        if (Kor() < KorHatar) {  
            PontSzam = viselkedesPont * Kor() + szepsegPont * (KorHatar - Kor());  
        } else {  
            PontSzam = 0;  
        }  
    }  
  
    // a ToString() így majd kisbetűsen kiírja az osztályneveket is.  
    public override string ToString() {  
        return RajtSzam + ". " + Nev + " nevű "  
            + this.GetType().Name.ToLower()  
            + " pontszáma: " + PontSzam + " pont";  
    }  
}
```

```

class Kutya : Allat {

    public int GazdaViszonyPont { get; private set; }
    public bool KapottViszonyPontot { get; private set; }

    public Kutya(int rajtSzam, string nev, int szulEv) :
        base(rajtSzam, nev, szulEv) {
    }

    public void ViszonyPontozas(int gazdaViszonyPont) {
        this.GazdaViszonyPont = gazdaViszonyPont;
        KapottViszonyPontot = true;
    }

    public override void Pontozzak(int szepsegPont, int viselkedesPont) {
        if (KapottViszonyPontot) {
            base.Pontozzak(szepsegPont, viselkedesPont);
            this.PontSzam += GazdaViszonyPont;
        }
    }
}

class Macska : Allat {

    public bool VanMacskaSzallitoDoboz { get; set; }

    public Macska(int rajtSzam, string nev, int szulEv, bool vanMacskaSzallitoDoboz) :
        base(rajtSzam, nev, szulEv) {
        this.VanMacskaSzallitoDoboz = vanMacskaSzallitoDoboz;
    }

    public override void Pontozzak(int szepsegPont, int viselkedesPont) {
        if (VanMacskaSzallitoDoboz) {
            base.Pontozzak(szepsegPont, viselkedesPont);
        }
    }
}

```

A vezérlés hajszál pontosan ugyanaz, mint a másik megoldás esetén.

4.2.8. Járműpark

Látva, hogy milyen ügyesen haladunk a programozás tanulásában, egyik barátunk, Jeromos, beavat a terveibe, és megkér, hogy írjuk meg az induló vállalkozása adminisztrálását végző programot. Úgy tervezí, hogy járműparkot akar üzemeltetni, egyelőre buszokat és teherautókat tud szerezni, ezeket akarja bérbe adni, de később nem kizárt, hogy esetleg másfajta járművel is bővíti a vállalkozását. Egy jókora telepet is talált hozzá, vagyis elvileg bármikor bővítheti a rendelkezésére álló járműparkot.



Ez bizony érdekes feladat, és határozottan jólesik, hogy a barátunk ennyire megbízik a tudásunkban. Ne hagyjuk cserben, és oldjuk meg a feladatot. Természetesen nem kezdünk bele húbelebalázs módjára, hanem tervezzük meg. Nagy esély van rá, hogy elkészülése után többször is módosítani kell majd a programot, és elsősorban nem azért, mert esetleg még nem annyira megbízható a tudásunk, és később jut eszünkbe egy-egy jobb ötlet (ilyen is előfordulhat), hanem azért, mert a barátunk igénye is változhat. Eleve avval kezdte, hogy ha jól működik a vállalkozása, akkor később más járműveket is beszerez. Vagyis úgy kell megírnunk a programot, hogy később könnyen módosítható, bővíthető legyen.

Először azt járjuk körbe, hogy mi a közös a buszokban és a teherautókban, sőt esetleges egyéb járművekben is. Persze, hogy az, hogy mindenki járművek ☺.

Már látszik, hogy a jármű fogalmából, azaz a **Jarmu** osztályból kell kiindulnunk. Először tehát fogalmazzuk meg az evvel kapcsolatos elvárásainkat:

Mindegyiket jellemzi a gyártási éve, és mindegyiknek van rendszáma. A rendszámot viszont akár ki is lehet cserálni, vagyis ez nem alkalmas a járművek egyedi azonosítására. Erre megfelelő lenne az alvázszám, de az is lehet, hogy a könyvelés során mi magunk adunk egyedi azonosítókat a járműveknek. Mivel ez az általánosabb elköpzelés, maradjunk ennél. (Ha később mégis az alvázszám mellett döntenénk, akkor csak az adatfájl tartalmát kell átírnunk, semmi más.)

Mindegyiknél fontos a fogyasztás, de mivel bonyolítaná a bérletdíj kiszámítását, ezért a barátunk úgy képzeli, hogy utanként megsaccolt fogyasztással számol majd. Mivel most használt járművekkel indít, ezért már mindegyik esetében ismeri ezt a fogyasztási értéket – amely természetesen utanként is változhat –, de nem kizárt, hogy valaha vadonatúj járműveket is vesz, amelyeknél ezt még nem lehet tudni.

Járművek esetén fontos tudnunk azt, hogy életük során összesen hány km-t futottak. Ennek értéke nyilván minden egyes fuvarozáskor növekszik.

Barátunk több összetevő alapján akarja kiszámolni a bérletdíjat. Azt gondolja, hogy minden egyes jármű esetén lesz egy egységes alapdíj, amelyhez hozzáadódik majd az aktuális út költségének egy szintén egységes haszonkulccsal növelt értéke. Az aktuális út költsége az aktuálisan megtett út hossza, az aktuális benzinár és a jármű fogyasztása alapján számolható.

Természetesen egy jármű csak akkor adható bérbe, ha épp szabad.

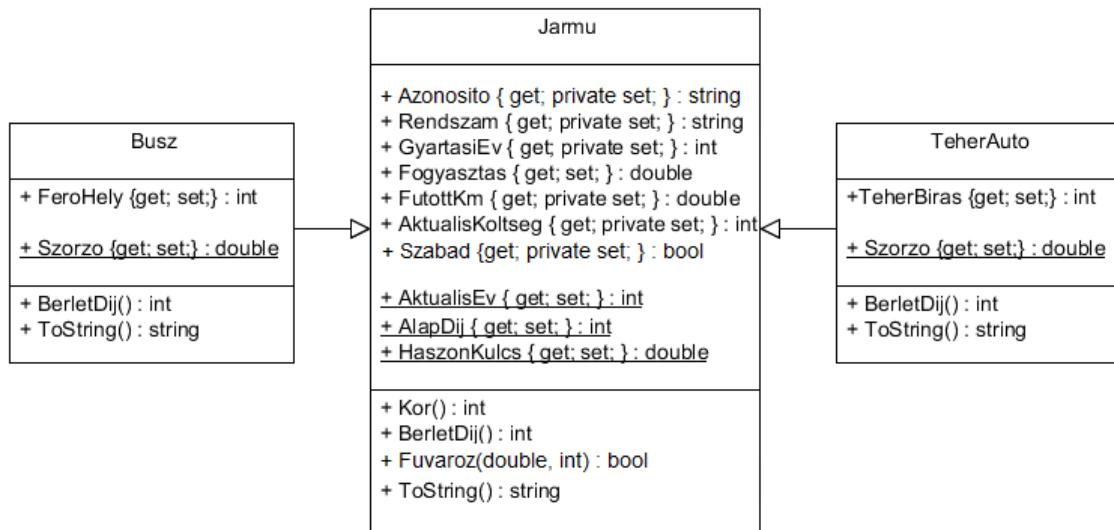


Jeromos úgy képzeli, hogy a **buszok** bérletdíját még a férőhelyek számától is függővé teszi, vagyis az előzőek alapján kiszámított bérletdíjhoz még hozzáadja a férőhelyek számának valahányszorosát. Úgy gondolja, hogy ennek a szorzónak az értékét is egységesen kezeli az összes busz esetén.

A **teherautók** bérbeadásakor pedig a jármű teherbírását akarja figyelembe venni, és a járművekhez tartozó bérletdíjhoz még hozzáadja a teherbírás valahányszorosát. Úgy tervez, hogy ennek a szorzónak az értékét is egységesen kezeli az összes teherautó esetén.



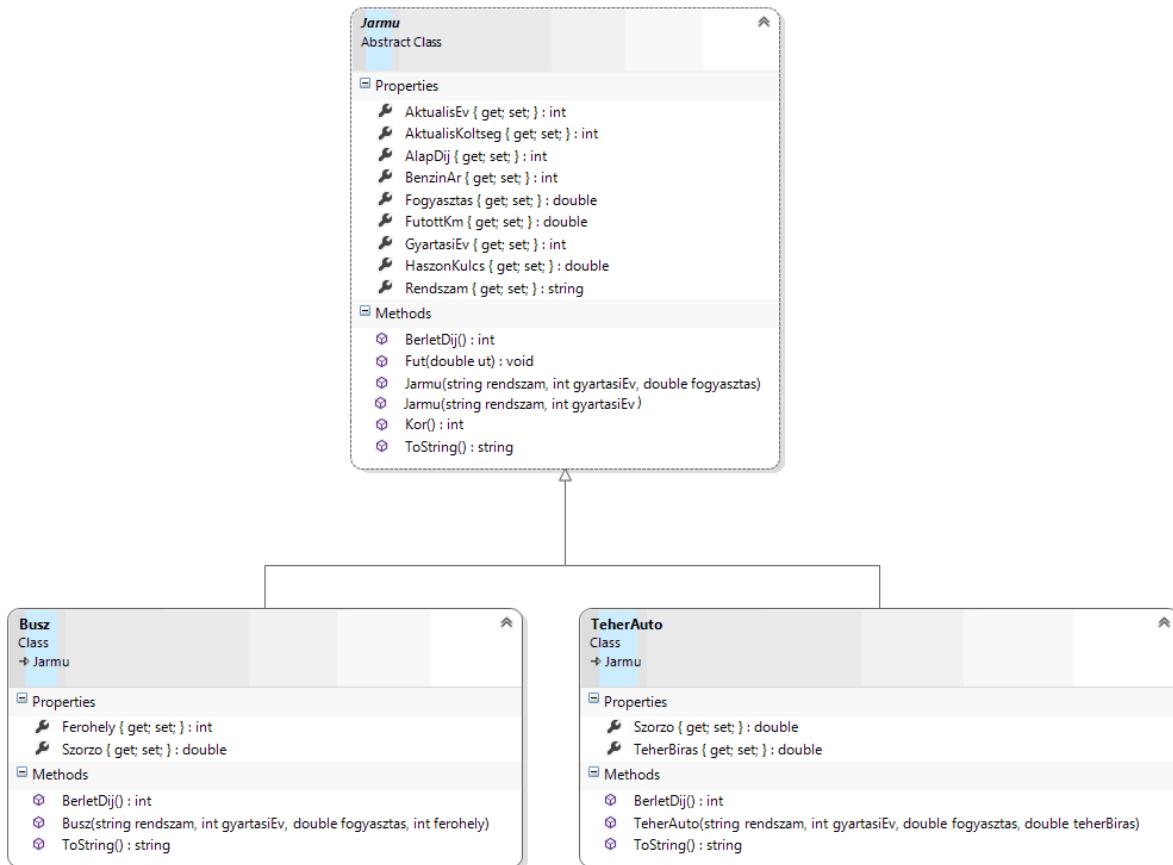
Úgy tűnik, végiggondoltuk az alapokat, a leírtak alapján próbáljuk meg létrehozni a feladat UML ábráját!



Vagyis nyilvánvaló, hogy a közös tulajdonságokat nem kell duplán leírni, ezek a közös ősosztályba kerülhetnek. Sőt mi több, ebből az osztályból esetleg további járműveket lehet származtatni (pl. személyautót, hajót stb.)

Ugyanakkor azt is végiggondolhatjuk, hogy hiszen egyetlen „ jármű” sem létezik, csak busz, teherautó, személyautó stb. van. Azaz a **Jarmu** osztályból egyetlen példányt sem hozunk létre, tehát kezelhetjük absztrakt osztályként is, vagyis olyan osztály, amelyből nem lehet példányt létrehozni, és elsődleges célja az, hogy egyéb osztályokat származtassunk belőle.

Ez a fajta megközelítés látható – az előzőhöz egyébként nagyon hasonló, csak – a Visual Studio által generált UML ábrán:



Ezek után lássuk az osztályok kódját:

```

abstract class Jarmu {

    // tulajdonságok (Properties)

    public string Azonosito { get; private set; }
    public string Rendszam { get; set; }
    public int GyartasiEv { get; private set; }
    public double Fogyasztas { get; set; }

    public double FutottKm { get; private set; }
    public int AktualisKoltseg { get; private set; }
    public bool Szabad { get; private set; }

    public static int AktualisEv { get; set; }
    public static int AlapDij { get; set; }
    public static double HaszonKulcs { get; set; }
}
  
```

```

// konstruktor (ha már ismerjük a fogyasztást)
public Jarmu(string azonosito, string rendszam, int gyartasiEv, double fogyasztas) {
    this.Azonosito = azonosito;
    this.Rendszam = rendszam;
    this.GyartasiEv = gyartasiEv;
    this.Fogyasztas = fogyasztas;
    this.Szabad = true;
}

// konstruktor (ha még nem ismerjük a fogyasztást, pl. vadonatúj a jármű)
public Jarmu(string azonosito, string rendszam, int gyartasiEv) {
    this.Azonosito = azonosito;
    this.Rendszam = rendszam;
    this.GyartasiEv = gyartasiEv;
    this.Szabad = true;
}

// azért nem a Foglalt tulajdonságot vezettük be, hogy ezt az értékadást
// is megmutathassuk.

// metódusok

/// <summary>
/// Kiszámolja a jármű korát.
/// </summary>
/// <returns></returns>

public int Kor() {
    return AktualisEv - GyartasiEv;
}

/// <summary>
/// A metódus paraméterében lévő értékkel növeli az eddig megtett kilométerek számát,
/// és kiszámolja az aktuális út költségét.
/// Beállítja, hogy foglalt (nem szabad)
/// </summary>
/// <param name="ut"></param>

// Ha bool típusúként kezeljük, akkor azt is tudjuk figyelni, hogy megvalósult-e a fuvar.
// Erre valószínűleg csak akkor "jövünk rá", amikor már a vezérlés részt is
// végiggondoltuk.
public bool Fuvaroz(double ut, int benzinAr) {
    if (Szabad) {
        FutottKm += ut;
        AktualisKoltseg = (int)(benzinAr * ut * Fogyasztas / 100);
        Szabad = false;
        return true;
    }
    return false;
}

/// <summary>
/// Kiszámolja az alap bérletdíjat.
/// </summary>
/// <returns></returns>
public virtual int BerletDij() {
    return (int)(AlapDij + AktualisKoltseg + AktualisKoltseg * HaszonKulcs / 100);
}

```

```

/// <summary>
/// Beállítja, hogy szabad
/// </summary>

public void Vegzett() {
    Szabad = true;
}

```

A `ToString()` metódus megvalósítására több alternatívát is mutatunk: az első egy egyszerű string-konkatenáció, a második kettő felhasználja a `String` osztály `Format()` metódusának szolgáltatásait.

```

public override string ToString() {
    return "\nA " + this.GetType().Name.ToLower() +
        " azonosítója: " + Azonosito +
        "\nrendszáma: " + Rendszam +
        "\n      kora: " + Kor() + " év" +
        "\n      fogyasztása: " + Fogyasztas + " l/100 km" +
        "\n      a km-óra állása: " + FutottKm + " km";
}

public override string ToString() {
    string nev = this.GetType().Name.ToLower();
    return string.Format("\nA {0,-10} \nazonosítója: {1,3} " +
        "\nrendszáma: {2,-8} " +
        "\n\tkora: {3,3} év" +
        "\n\tfogyasztása: {4,3:00.0} liter/100 km   " +
        "\n\ta km-óra állása: {5,8:00.00} km",
        nev, Azonosito, Rendszam,
        Kor(), Fogyasztas, FutottKm);
}

public override string ToString() {
    Object[] obj = { this.GetType().Name.ToLower(), Azonosito, Rendszam,
        Kor(), Fogyasztas, FutottKm };
    return string.Format("\nA {0,-10} \nazonosítója: {1,3} " +
        "\nrendszáma: {2,-8} " +
        "\n\tkora: {3,3} év" +
        "\n\tfogyasztása: {4,3:00.0} liter/100 km   " +
        "\n\ta km-óra állása: {5,8:00.00} km", obj);
}

```

Az utód osztályok:

```

class Busz : Jarmu {

    // tulajdonságok
    public int Ferohely { get; private set; }
    public static double Szorzo { get; set; }

    // konstruktorok
    public Busz(string azonosito, string rendszam, int gyartasiEv,
                double fogyasztas, int ferohely) :
        base(azonosito, rendszam, gyartasiEv, fogyasztas) {
            this.Ferohely = ferohely;
    }

    public Busz(string azonosito, string rendszam, int gyartasiEv, int ferohely) :
        base(azonosito, rendszam, gyartasiEv) {
            this.Ferohely = ferohely;
    }

    //metódusok

    // Kiszámolja a busz bérletdíját.

    public override int BerletDij() {
        return (int)(base.BerletDij() + Ferohely * Szorzo);
    }

    public override string ToString() {
        return base.ToString() +
            "\n\tFérőhelyek száma: " + Ferohely;
    }
}

class TeherAuto : Jarmu {

    // tulajdonságok
    public double TeherBiras { get; private set; }
    public static double Szorzo { get; set; }

    // konstruktor
    public TeherAuto(string azonosito, string rendszam, int gyartasiEv,
                     double fogyasztas, double teherBiras) :
        base(azonosito, rendszam, gyartasiEv, fogyasztas) {
            this.TeherBiras = teherBiras;
    }

    public TeherAuto(string azonosito, string rendszam, int gyartasiEv, double teherBiras) :
        base(azonosito, rendszam, gyartasiEv) {
            this.TeherBiras = teherBiras;
    }

    //metódusok

    // Kiszámolja a teherautó bérletdíját.

    public override int BerletDij(){
        return (int)(base.BerletDij() + TeherBiras * Szorzo);
    }

    public override string ToString() {
        return base.ToString() +
            "\n\tTeherbírás: " + TeherBiras + " tonna";
    }
}

```

Működtessük az elkészült osztályokat, vagyis

- Olvassuk be a szükséges adatokat egy adatfájlból (a statikusakat konstansként is beállíthatja).
- Működtessük a járműparkot. Ez most ennyit jelent: Véletlen sokszor ismételjük meg: Válasszunk ki véletlenül egy járművet. Ha ez képes fuvarozni, akkor számoljuk a fuvarok számát, a cég összes bevételét (véletlen benzinár, véletlen úthossz esetén) és az összes költségét. Véletlenszerűen határozzuk meg azt a járművet is, amelyik épp végzett a fuvarral.
- Készítsünk statisztikákat, azaz határozzuk meg a járművek átlag-életkorát, azt, hogy melyik jármű(vek) futott(ak) a legtöbbet, illetve rendezzük őket fogyasztás szerint növekvő sorrendbe.

Mielőtt ténylegesen megírnánk a program vezérlés részét, beszéljünk meg valamit.

Definiálunk egy `Jarmu` típusú példányokból álló, `jarmuvek` nevű listát, és tegyük bele vegyesen néhány `busz` és néhány `teherAuto` példányt. Gondoljuk végig, mi történik az alábbi kód részlet hatására:

```
for (int i = 0; i < jarmuvek.Length; i++) {  
    Console.WriteLine(jarmuvek[i].Rendszam + " bérletdíja " +  
                      jarmuvek[i].BerletDij() + " Ft.");  
}
```

Mivel a `Jarmu` osztályban definiáltuk a `Rendszam` tulajdonságot is és a `BerletDij()` metódust is, ezért ez a kód részlet működik. De milyen bérletdíjakat ír ki? Természetesen busz példányok esetén a buszoknak megfelelő díjat, teherautók esetén pedig a teherautóknak megfelelőt, vagyis buszok esetén a férőhelyek számával számol, teherautók esetén a teherbírással.

Biztos, hogy ez „természetes”? Ha igennel válaszol, akkor vagy már ismeri a fontosabb OOP fogalmakat, vagy egyáltalán nem érdekli az egész. Ez utóbbi teljesen elköpzelhetetlen ☺, és talán senkinek sem árt, ha meg is beszéljük, hogy mi miért működik.

Kérdés tehát, hogy honnan tudja a program, hogy az i-edik jármű esetén melyik `BerletDij()` metódussal kell számolnia? A férőhelyek számát vagy a teherbírást kell figyelembe vennie? Ez fordítási időben nem is derül ki, hiszen a fordító csak annyit lát, hogy `Jarmu` típusú példányok `BerletDij()` metódusát kell meghívni. Az, hogy a konkrét példány milyen fajta jármű, busz-e vagy teherautó, és milyen módon írtuk felül a metódust, csak futáskor derül ki, de ekkor a futtató környezet már tudja, hogy melyik metódust kell alkalmaznia. Ezt a kései felismerést „kései kötés”-nek (late binding) nevezik. Ez a **polimorfizmus** (többalakúság) elve, amely azt jelenti, hogy egy ōs típusra deklarált változó helyére bár melyik Utód típusú objektumot illeszthetjük, a program helyesen fut, akkor is, ha az utód felülírta az ōs meghívott metódusát.

Térjünk vissza az eredeti feladatra. Persze, egyáltalán nem volt haszontalan az előző kitérő, hiszen ennek során az eredeti feladattal kapcsolatos kérdéseket beszéltek meg, csak most tovább kell lépnünk.

A megoldás – remélhetőleg – már rutinnak számít, egyetlen dolgot kell megbeszélnünk, a beolvasandó fájl szerkezetét. Barátunk úgy képzeli, hogy a könnyebb olvashatóság kedvéért a fájlban legyen feltüntetve, hogy épp busz vagy teherautó adatai következnek, utána pedig soronként egyetlen adatot tartalmazva ezek szerepeljenek: a jármű rendszáma, gyártási éve, 100 km-enkénti ismert fogyasztása és buszok esetén a szállítható személyek száma, teherautó esetén pedig a teherbírás.

Úgy képzeli, hogy a járművek a beolvasás sorrendjében kapják meg az azonosítójukat: a jármű fajtájának kezdőbetűje (B/T), plusz a beolvasás sorszáma, 1-ről indulva.

A továbbiakban közölt kód nem igényel magyarázatot:

busz
BAR-123
2012
20
40
teherautó
CAD-312
2010
10
1,5
teherautó
LKA-412
2008
12
4,5
busz
DRS-374
2010
13
30
busz
GHK-517
2009
15
35

```
class Vezerles {

    private List<Jarmu> jarmuvek = new List<Jarmu>();
    private string BUSZ = "busz";
    private string TEHER_AUTO = "teherautó";

    public void Indit() {
        StatikusBeallitas();
        AdatBevitel();
        Kiir("A regisztrált járművek: ");
        Mukodtet();
        Kiir("\nA működés utáni állapot: ");
        AtlagKor();
        LegtobbKilometer();
        Rendez();
    }

    private void StatikusBeallitas() {
        // Statikus adatok beállítása, természetesen be is lehetne olvasni.
        Jarmu.AktualisEv = 2015;
        Jarmu.AlapDij = 1000;
        Jarmu.HaszonKulcs = 10;

        Busz.Szorzo = 15;
        TeherAuto.Szorzo = 8.5;
    }
}
```

```

private void AdatBevitel() {
    string tipus, rendszam, azonosito;
    int gyartEv, ferohely;
    double fogyasztas, teherbiras;

    StreamReader olvasoCsatorna = new StreamReader("jarmuvek.txt");

    int sorszam = 1;

    while (!olvasoCsatorna.EndOfStream) {

        tipus = olvasoCsatorna.ReadLine();
        Console.WriteLine(tipus);
        rendszam = olvasoCsatorna.ReadLine();
        gyartEv = int.Parse(olvasoCsatorna.ReadLine());
        fogyasztas = double.Parse(olvasoCsatorna.ReadLine());
        azonosito = tipus.Substring(0, 1).ToUpper() + sorszam;

        if (tipus == BUSZ) {
            ferohely = int.Parse(olvasoCsatorna.ReadLine());
            jarmuvek.Add(new Busz(azonosito, rendszam, gyartEv,
                                  fogyasztas, ferohely));
        }
        else if (tipus == TEHER_AUTO) {
            teherbiras = double.Parse(olvasoCsatorna.ReadLine());
            jarmuvek.Add(new TeherAuto(azonosito, rendszam, gyartEv,
                                       fogyasztas, teherbiras));
        }
        sorszam++;
    }
    olvasoCsatorna.Close();
}

private void Kiir(string cim) {
    Console.WriteLine(cim);
    foreach (Jarmu jarmu in jarmuvek) {
        Console.WriteLine(jarmu);
    }
}

/// <summary>
/// Véletlen sokszor ismételjük meg:
/// Válasszunk ki véletlenül egy járművet.
/// Ha ez képes fuvarozni, akkor számoljuk ki
/// a fuvarok számát,
/// a cégek összes bevételét (véletlen benzinár, véletlen úthossz)
/// és az összes költségét.
/// Egy véletlenszerűen választott jármű végez a fuvarral.
/// </summary>
private void Mukodtet() {
    // Összetettebb változat
    int osszKoltseg = 0, osszBevetel = 0;
}

```

```

Random rand = new Random();
int alsoBenzinAr = 400, felsoBenzinar = 420;
double utMax = 300;
int mukodesHatar = 200;
int jarmuIndex;

Jarmu jarmu;
int fuvarSzam = 0;

for (int i = 0; i < (int)rand.Next(mukodesHatar); i++) {
    jarmuIndex = rand.Next(jarmuvek.Count);
    jarmu = jarmuvek[jarmuIndex];
    if (jarmu.Fuvaroz(rand.NextDouble() * utMax,
                       rand.Next(alsoBenzinAr, felsoBenzinar))) {
        // így most akár 0 km-t is lehet
        fuvarSzam++;
        Console.WriteLine("\nAz idnuló jármű rendszáma: " + jarmu.Rendszam
                        + "\nAz aktuális fuvarozási költség: " + jarmu.AktualisKoltseg + " Ft."
                        + "\nAz aktuális bevétel: " + jarmu.BerletDij() + " Ft.");
        osszBevetel += jarmu.BerletDij();
        osszKoltseg += jarmu.AktualisKoltseg;
    }

    jarmuIndex = rand.Next(jarmuvek.Count);
    jarmuvek[jarmuIndex].Vegzett();
}

Console.WriteLine("\n\nA cégl teljes költsége: " + osszKoltseg + " Ft."
                  + "\n\nTeljes bevétele: " + osszBevetel + " Ft."
                  + "\n\nHaszna: " + (osszBevetel - osszKoltseg) + "Ft.");
Console.WriteLine("\nA fuvarok száma: " + fuvarSzam);
}

private void AtlagKor() {
    double osszKor = 0;
    int darab = 0;
    foreach (Jarmu jarmu in jarmuvek) {
        osszKor += jarmu.Kor();
        darab++;
    }

    if (darab > 0) {
        Console.WriteLine("\nA járművek átlag kora: " + osszKor / darab + " év.");
    }
    else {
        Console.WriteLine("Nincsenek járművek.");
    }
}

```

```

private void LegtobbKilometer() {
    double max = jarmuvek[0].FutottKm;
    foreach (Jarmu jarmu in jarmuvek) {
        if (jarmu.FutottKm > max) {
            max = jarmu.FutottKm;
        }
    }
    Console.WriteLine("\nA legtöbbet futott jármű(vek) {0: 000.00} km-rel:", max);
    foreach (Jarmu jarmu in jarmuvek) {
        if (jarmu.FutottKm == max) {
            Console.WriteLine(jarmu.Rendszam);
        }
    }
}

private void Rendez() {
    Jarmu temp;

    for (int i = 0; (i < jarmuvek.Count-1); i++) {
        for (int j = i + 1; (j < jarmuvek.Count); j++) {
            if (jarmuvek[i].Fogyasztas > jarmuvek[j].Fogyasztas) {
                temp = jarmuvek[i];
                jarmuvek[i] = jarmuvek[j];
                jarmuvek[j] = temp;
            }
        }
    }

    Console.WriteLine("\nA járművek fogyasztás szerint rendezve: ");

    foreach (Jarmu jarmu in jarmuvek) {
        Console.WriteLine("{0,-10} {1:00.0} liter / 100 km.",
                        jarmu.Rendszam, jarmu.Fogyasztas);
    }
}

```

Hurrá, sikerült, megcsináltuk! ☺ Boldog lehet Ön is és Jeromos is.

Ámde amilyen nagy volt a kezdeti eupória, barátunk arcán újabban egyre több zavart lehet látni. Végül kibökte, mi zavarja. Elárulta, hogy múltkoriban beszaladt hozzá a szomszéd kisgyerek, Zakariás, és mikor meglátta a számítógépen futó programot, azonnal közölte, hogy az ő apukája ennél sokkal szébbet tud. (Ez igaz is, az apukája itt végzett a Pollackon. ☺)

A gyerek elmesélése alapján már Jeromos is el tudja képzelni, hogy nagyjából mit szeretne. Valami ilyesmit:

Így induljon az alkalmazás:

Azaz legyen valami érdekes kezdőoldala, és különböző fülek segítségével lehessen elérni az egyes részfeladatokat, mint amilyen a buszok és teherautók bérbeadása, illetve legyen egy oldal a cég könyvelése számára is.

Úgy képzeli, hogy induláskor valahogy így nézzen ki a három kezdőoldal:



Hogy az adatok a program indulásakor automatikusan töltődjenek-e be, vagy ki lehessen választani, hogy melyik adatfájlban (vagy esetleg adatbázisban) vannak az adatok, ezt egyelőre még nem tudta megmondani, de talán pillanatnyilag nem is érdekes. (Az automatikus betöltés már eddig is működött. ☺)

Látszik, hogy a buszokra és teherautókra vonatkozó felületek nagyjából egyformák, ezért elég a buszok esetében megfogalmazni az elvárásokat.

A felület bal oldalán vannak felsorolva a bérelhető buszok (a buszok rendszáma olvasható a listában).

Alatta be lehet állítani az aktuális benzinárat, illetve a megteendő út hosszát. Ezeket a megrendelés előtt

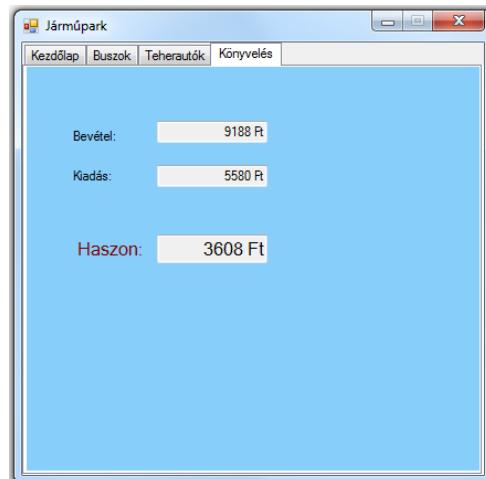
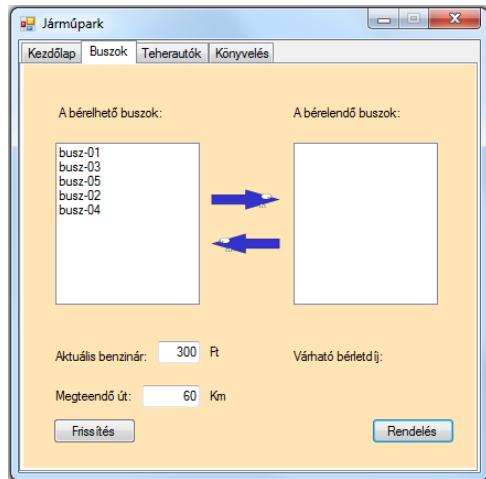
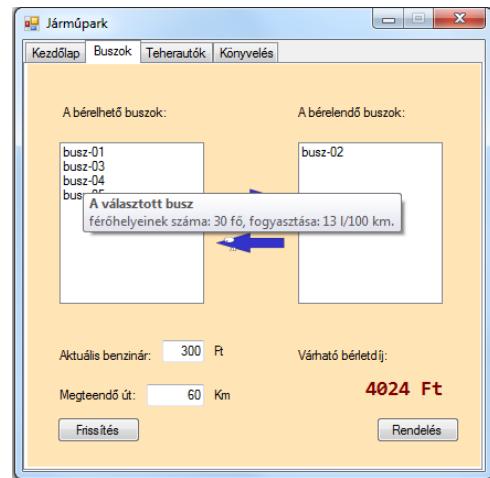
még bármikor lehet módosítani, a Frissítés gomb hatására mindenkor a legfrissebb adatokkal számol majd a program.

Hogy könnyebb legyen a választás, a listában lévő rendszámokra kattintva (vagy föléjük mozgatva az egeret) kapunk egy kis információt az aktuális buszról, vagyis jelenjen meg, hogy hány férőhelyes és kb. mennyi a fogyasztása.

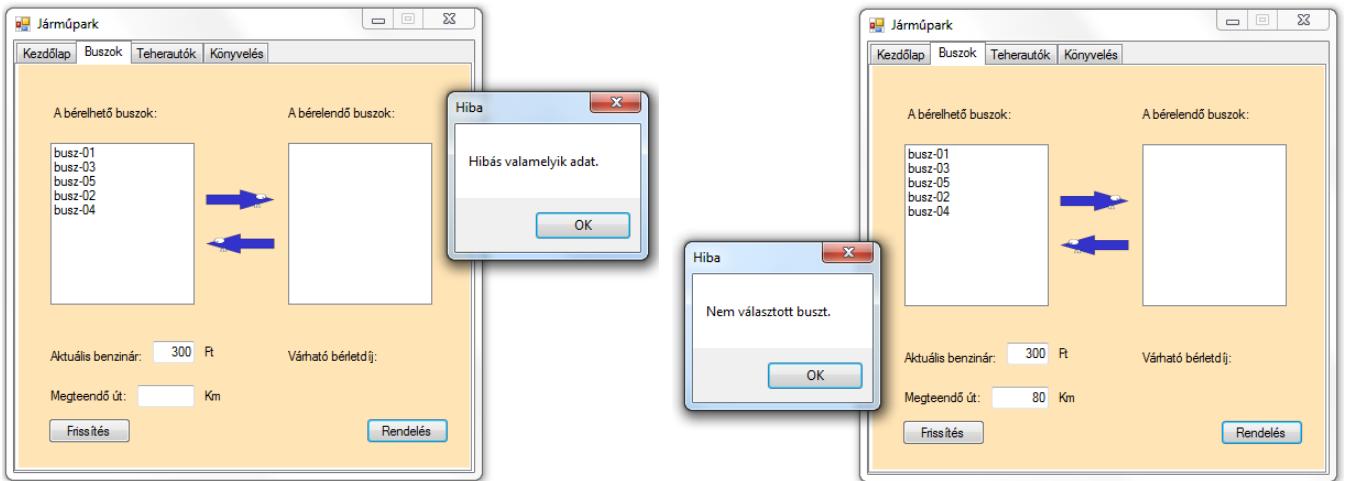
A jobbra mutató nyílra kattintva kerüljön be a kiválasztott rendszám a bérelhető buszok listájába, és a lista alján tudjuk olvasni a várható bérletdíj értékét is. Természetesen, ha egynél több buszt választunk, akkor a várható bérletdíj az egyes bérletdíjak összege.

Arra is legyen mód, hogy a megrendelő meggondol-hassa magát, vagyis ha pl. sokallja a fizetnivalót, akkor a balra mutató nyílra kattintva visszatehesse a kiválasztott rendszámot a bérelhető buszok listájába. Ekkor természetesen a várható bérletdíj értéke is csökken.

Ha végül úgy dönt az ügyfél, hogy megrendeli a kiválasztott buszt vagy buszokat, akkor a Rendelés gombra kattintva ezt meg is teheti, ennek hatására a megrendelő felület visszaáll alap-állapotba, a könyvelési oldalon pedig elkönyvelik a bevételt, költséget és a hasznöt.



Természetesen az is fontos, hogy esetleges hibás adat esetén, vagy ha elfelejtünk kitölteni valamit, akkor hibajelzést kapunk:



Persze, egyértelműen látszik az elköpzeléséből, hogy nyilván nem Ő könyvel, mert a hatóság bizony nem elég szik meg csak a haszon kiszámításával, ennél jóval precízebb kimutatások kellenek, feltüntetve a bérlet dátumát és egyéb adatokat, de egyelőre maradjunk ennyinél.

Jeromos barátunk feladta a leckét, de ne ijedjen meg, remélhetőleg nem kellenek hozzá évek, hanem néhány hét alatt már meg is tudja tanulni az ehhez szükséges tudnivalókat.

Sok sikert hozzá! 😊